2.1.2 Operations

Having described the relationship between data structures, abstract data types, and operations, consider the generic structure of an operation. An *operation*, in its most primitive form, is a simple request/reply interaction. The interaction takes the following form:

- The operation is *invoked*. An invocation consists of:
  - an *operation number*, which uniquely identifies the operation to be performed;
  - an arbitrarily complex *argument*;
  - an *invocation identifier*, which is used to distinguish this invocation from other previous invocations; and,
  - possibly a *linked invocation identifier*, which is used to indicate that this invocation results during the processing of a previous invocation.
- If the operation succeeds, then a *result* may be returned. A result consists of:
  - an invocation identifier corresponding to the operation which succeeded; and,
  - (possibly) an arbitrarily complex *result*.
- If the operation fails, then an *error* is returned. An error consists of:
  - an invocation identifier corresponding to the operation which failed;
  - an *error number*, which uniquely identifies the error that occurred; and,
  - (possibly) an arbitrarily complex *parameter*.
- If the operation was not performed for some reason (e.g., an unknown operation number, mistyped arguments for the operation, and so on), then a *reject* is returned. A reject consists of:
  - an invocation identifier corresponding to the operation which was performed (in degenerate cases this information may not be available); and,

- a *reason*, which describes, in general terms, the rejection which occurred.

Note that in all cases, operations are seen as being *total*. This means that for any given operation, the normal outcome (the result) and all exception outcomes (the errors) are well-defined and mutually exclusive (unambiguously distinguished).

2.1.3 Associations

Thus far, we have implicitly assumed some communications relationship (i.e., a *binding*) between the entity invoking operations, the *invoker*, and the entity performing them, *performer*. It is necessary formalize these notions somewhat.

An *association* is a binding between two entities, which are referred to as the *initiator* and the *responder*. Although this sounds straight-forward, there is actually one level of indirection present: the initiator does not directly select a responder for binding, but rather selects a *service* that it wishes to use. During this process a responder is located and the binding established.

The binding process is actually two-step: first, the initiator determines which service it requires and asks to have this service mapped onto the entities available on the network; second, based on the initiator's communications requirements, an association will be bound to one of those entities, which becomes the responder.

Note that there is no requirement that the initiator of an association be the entity which requests operations to be invoked (i.e., the initiator need not be an invoker), nor that the responder of an association be the entity which performs the operations (i.e., the responder need not be a performer).

2.2 Design Guidelines

Although the characteristics of operations will vary between applications, there are certain guidelines which should be of universal interest.

2.2.1 Reliability Characteristics

When executing an operation in a distributed environment, there is always an element of uncertainity with regards to the occurrence of remote events, particularly when failures occur. One classification of reliability might be as follows: for a given invocation, a guarantee is given that it occurs *exactly once*, *at least once*, or *at most once*.

The first two guarantees require an end-to-end confirmation, with the proviso that the semantics of *exactly once* require a recovery scheme in the event of failures. When an operation has the semantics of *at least once*, it is called *idempotent*. These are operations which read information, but do not modify the remote state.

In practice, implementing these semantics can be surprisingly straightforward, given the judicious use of invocation identifiers:

exactly once: The invoker repeatedly requests the operation (using the same invocation identifier) until either a confirmation (result or error) is received or a rejection of "duplicate operation" is received. The performer keeps track of the invocation identifiers of all performed operations requested by an entity from an epoch date. If the performer finds an invocation identifier being repeated, rather than perform the operation, it rejects the operation as a "duplicate operation".

at least once: The invoker repeatedly requests the operation (with any invocation identifier) until a confirmation (result or error) is received. The performer has no state information regarding previously used invocation identifiers.

at most once: The invoker requests the operation exactly once with any invocation identifier. The performer has no state information regarding previously used invocation identifiers.

2.2.2 Keeping Total Operations Total

An operation may be thought of as *confirmed* when either a result or error is returned by the performer. However, under the OSI framework for remote operations, it is possible to have an operation which may only return a result,

or may only return errors. This is a subtle point, but one worth emphasizing: in the past some operations have been defined which on success do not return a result, this lead to ambiguity in as much as the invoker could not determine the "correct" time to terminate the association, since "silence" on the part of the performer could mean that the operation completed successfully or that the operation was still in progress and an error return might be forthcoming. In keeping with the philosophy of total operations, it is important to have an operation return a result, even if the information is `NULL`.

2.3 For Further Reading

The first half of [ECM85] is the most authoritative reference on using remote operations. In particular, Section 4.4 presents a simple template methodology for using remote operations. As mentioned earlier, Section 5 and beyond have been obsoleted by work in ISO and CCITT.

In as much as the standards from these bodies deal with the remote operations service and protocol, they are not particularly germane here. The ISO work on remote operations is presented in [ISO88a] and [ISO88b]. The corresponding CCITT work is described in [CCI88a] and [CCI88b]. The native interface to these services are discussed in Chapter 3 of *Volume One*.

Chapter 3

A Discipline for Meal Preparation

The previous chapter presented a methodology for building distributed applications, based on two key aspects: *abstract data types*, and *remote operations*. During the exposition of these abstract terms, they were related to their more concrete realization in an OSI architecture, namely: abstract syntax, association control, and remote operations.

In the chapters that follow, this volume will introduce the utensils (programming tools) and various recipes (boilerplate routines) which can be used to cook-up OSI applications. Future editions of this volume will no doubt detail how one can prepare either fast food, nouvelle cuisine, or gourmet cooking. Given the raw power of remote operations in OSI, the cooking analogy is actually quite appropriate: some features of the remote operations service will be restricted in order to make the programming work more digestable!

In this chapter, the steps described in the following chapters are introduced and placed in perspective. Throughout the remaining chapters, an extended example is presented to illustrate, step-by-step, the use of remote operations.

3.1 Defining A New Service

A collection of remote operations forms *only* a partial definition of a service. Other parts of this definition include naming and addressing information.

Chapter 10 of *Volume One* describes fully how a service is defined in these terms. Rather than reproduce that chapter in this volume, let us consider the essential points. At first reading, this seems complicated. In point of fact, the first definition of a service is challenging, defining later services is straight-forward and only marginally tedious!

Here are the ones we are interested in:

abstract syntax: this describes the data structures being exchanged by the service;

application context name: this describes the protocol being used by the service;

application-entity information: this uniquely names an entity in the network;

presentation address: this locates an entity in the network; and,

local program: this identifies the program on the local system which implements the service.

We now begin the example which will be repeatedly extended in the chapters which follow. Let us suppose that we are defining the *ISODE Image Service*, which offers a simple image database service.

Although Chapter 10 of *Volume One* defines several approaches, we will take the "standard" approach for our example:

abstract syntax: defined in the *isobjects*(5) file as:

```
"local service pci"      1.17.2.n.1
```

If we select `n` as the lowest unassigned number in the `1.17.2` subtree, e.g., 7, we might have:

```
"isode image pci"      1.17.2.7.1
```

application context name: defined in the *isobjects*(5) file as:

```
"local service"          1.17.2.n.2
```

Similarly, for a value of 7 for `n`, we have:

```
"isode image"          1.17.2.7.2
```

application-entity information/presentation address: defined with:

```
default servicestore    1.17.4.1.n  ""  ""  #p
```

in the *isoentities*(5) file (we say that `servicestore` is the "qualifier" portion of the service). If we select `p` as the lowest unassigned TSAP ID between `1024` and `2047` inclusive, e.g., `1040`, we might have:

```
default imagestore     1.17.4.1.7  ""  ""  #1040
```

local program: defined in the *isoservices*(5) file as:

```
"tsap/servicestore"    #p  program arg1 arg2 ... argN
```

If we select *ros.image* as the UNIX program implementing the *ISODE Image Service*, we might have something like:

```
"tsap/imagestore"    #1040  ros.image
```

By convention, the names of responder programs start with `"ros."`.

In future releases of the software, a facility might be added to determine most of this information directly from the formal definition of the service (i.e., from the remote operations module described momentarily).

The definition of this naming and addressing information is an administrative necessity. With this out of the way, let's consider the actual work that goes into building a distributed application which uses remote operations.

3.2 Defining A Remote Operations Module

A *remote operations module* defines the operations (and associated errors) along with the abstract syntax of the data structures exchanged by the service. The *rosy*(1) program reads a description of the module and produces the corresponding *C* stubs and definitions used in accessing the service.

In simple terms, what this means is that *rosy* will produce routines that the invoker calls that will request the operation by the performer.

3.3 Defining Concrete Data Structures

An *abstract syntax module* defines the data structures being exchanged by the service. In many senses, it is equivalent to a remote operations module with the definitions of the operations removed. The *posy*(1) program reads a description of the module and produces the corresponding *C* structure definitions along with an augmented module that is read by the *pepy* compiler. The *C* structure definitions are used by the invoker and the performer, the run-time environment is responsible for mapping these data structures to the abstract syntax and then for mapping the abstract syntax to the concrete syntax.

In simple terms, what this means is that the invoker and performer worry about only the native machine *C* structures, and the open systems interface is handled entirely automatically.

3.4 Building An Initiator

In general, an initiator might take one of two implementational forms:

interactive: the user runs a program and interactively directs the invocation of operations; or,

embedded: as a part of its running, the program automatically forms an association and invokes operations as required.

The choice of form is up to the implementor. Note that in both cases, the initiator is also acting as the invoker.

In order to speed development, complete boilerplate for an interactive initiator, along with partial boilerplate for an embedded initiator is provided.

3.5 Building A Responder

In general, a responder may also take on of two implementational forms:

single association: each time the service is requested, a new instantiation of the program implementing the service is executed (this is termed the *dynamic* approach); or,

multiple association: each time the service is requested, the request is given to a single, already executing, instantiation of the program which implements the service (this is termed the *static* approach).

Although Section 3.1 describes how to define addressing information for the dynamic approach, it is good practice to implement each responder to be capable of operating in either mode and then leaving the decision to the system administrator.

In order to speed development, complete boilerplate for a dual-approach responder is provided.

3.6 Putting It All Together

In addition to the skeletal information above, a set of consistent policies for file names, rules for *make*(1), and so on, are needed. These will be introduced at the appropriate time.

It is important to understand that this volume defines **one** possible discipline for using remote operations. There are, of course, **many** possible disciplines for accomplishing the same goal. (And some will most likely be vastly superior.) The particular discipline described in *The Applications Cookbook* was chosen primarily for its balance of relative simplicity and extensibility. For some developers, it will still be too under done; if this is the case, you are encouraged to build further cooked interfaces on top of these facilities.