An ELROS Primer

M.L. Branstetter
J.A. Guse
D.M. Nessett
L.C. Stanberry

"*Elros* Son of Eärendil and Elwing, who at the end of the First Age chose to be numbered among Men, and became the first King of Númenor (called *Tar-Minyatur*), living to a very great age." [**The Silmarillion**, J.R.R. Tolkien]

Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Acknowledgments

The design and implementation of a sophisticated system like ELROS generally involves a large number of contributors. We greatly appreciate and acknowledge the work and assistance of the following people without which we could not have brought ELROS into the world.

Robyne Sumpter worked with us for approximately a year on the implementation of the ELROS runtime system. She provided us with very valuable expertise in multi-tasking and communications interfaces. Robyne left the ELROS project to become a project leader in the storage group and we wish her all the best in her new responsibilities.

Rob Shaw spent about five months reworking the ELROS preprocessor parser, converting it to an enhanced C-based grammar. We didn't have much time to get to know Rob, but during his time with the project, we were very impressed with his abilities. We are sure that he will excel in his graduate studies in computer science and will miss his exuberance and strong work ethic.

Both Gary Long and Ashley Allen were early participants in the ELROS design effort. While their tenure with the project was short, several features of the ELROS embedded language are the result of their suggestions and contributions.

Mark Boolootian has recently joined the ELROS project to work on optimizing communications performance. We expect him to make major contributions to our work in the future.

We also thank John Fletcher and Dave Hardin for their helpful comments on earlier drafts of this primer.

Table Of Contents

Notice, Disclaimer, and Conditions of Use

ELROS is a preprocessor that accepts language statements embedded in C programs and produces C code to encode/decode ASN.1 as well as send and receive remote operations. The ELROS software is available for use under the following conditions. Permission to use, copy, modify and distribute the software and its documentation without fee or royalty is hereby granted, provided that you agree to comply with the copyright notice and following statements, including the disclaimer, and that the same appear on ALL copies of the software and documentation, including modifications that you make for internal use or for distribution.

1. ELROS is copyrighted by the University of California. Title to copyright to ELROS and to any material associated therewith shall at all times remain with the UNIVERSITY. ELROS was developed under Contract W-7405-ENG-48 between the University of California and the U.S. Department of Energy.

2. UNIVERSITY MAKES NO REPRESENTATION OR WARRANTIES, EXPRESS OR IMPLIED. By way of example, but not limitation, UNIVERSITY makes no representations or warranties of merchantability or fitness for any particular purpose or that the use of software components or documentation will not infringe any patents, copyrights, trademarks or other rights. UNIVERSITY shall not be liable for any direct, indirect or consequential damages with respect to any claim by USER or any third party on account of or arising from this Agreement or use of ELROS.

3. USER shall observe all applicable United States and foreign laws, regulations, rules and decrees with respect to the transfer of Program and related technical data to foreign countries, including without limitation, the US Foreign Corrupt Practices Act, The Export Administration Act, the US Anti-Boycott Laws and any regulations promulgated thereunder, including, but not limited to, the International Traffic in Arms regulations.

I. Introduction

There is increasing interest in the science and engineering disciplines to use distributed applications to improve computational performance; to couple general purpose computational systems with specialized hardware, such as visualization processors, archival storage and data acquisition equipment; and to migrate large programs from one computer architecture to another in a way that incrementally builds confidence in the correctness of the ported code. The fundamental support service for building distributed applications is remote procedure call (RPC), or in its more general form, remote operations (RO). This service is accessed by the applications programmer through a programming interface suitable for specifying remote service requests.

First generation programming interfaces for RPC and RO were based on the stub generator. The programmer provided a specification of a remote service procedure and its associated parameters to the stub generator, which produced a subroutine that an application program called to access the remote service. From the application programmer's point of view the subroutine was a surrogate of the remote procedure that implemented the remote service.

To implement the remote service, either the application programmer or someone else wrote a remote procedure, which was linked with a dispatcher[1] running on the remote system. When a remote request arrived at the dispatcher, it mapped the procedure identifier contained within the request into the corresponding local address of the procedure, called that procedure, and then returned the results of the call to the application which made the request.

Experience with the stub generator approach uncovered several deficiencies. Stub generators provide a natural way to express synchronous computations. However, they are clumsy at expressing asynchronous computations, since they provide the application programmer with a

[1] A dispatcher is a program loop that makes the appropriate system calls to receive communications from a remote correspondent and then parses the information contained therein to determine the appropriate actions to take.

synchronous interface to a remote service, i.e., a local subroutine call. Stub generators also are difficult to use. Development of a new remote service or modifications to an existing service requires not only modifying the application source code to provide for a new procedure call or new parameters to an existing call, but also the production of new stubs and linking them to the application and the dispatcher. When stubs and remote services are being developed and maintained by an individual different than the application programmer, this works reasonably well. When the application programmer is writing both the remote service call as well as the remote service procedure, a simpler description of the interactions between service caller and service provider is desirable. Finally, the stub generator approach requires the application programmer to deal with the underlying communications interface, passing the identifier of an open connection in the stub call. This requirement means the programmer must become familiar with two generally dissimilar interfaces rather than a single programming tool.

These considerations have led to the development of a second style of programming interface for remote service requests — the embedded language approach. Service requests are expressed in the form of a special programming language embedded in a conventional language, such as C. The embedded language statements are processed by a preprocessor, which translates these statements into statements of the conventional language. From the programmer's point-of-view, programming a distributed application is very similar to programming a stand-alone application. The conventional programming language is used to implement computations that are not distributed and the statements of the embedded language are used to express requests for remote service. Since the semantics of the embedded language are designed with distributed programming in mind, modes of operation not supported by the conventional language, such as asynchronous service requests, can be expressed in an intuitive and natural form. In addition, a simple coherent programming interface is provided that offers both communications primitives as well as primitives for remote service.

ELROS is an example of this new style of remote service interface. The application programmer embeds ELROS statements into C programs. The

ELROS preprocessor accepts these programs and produces the necessary C code to implement the semantics of the embedded statements.

This document provides an introduction to ELROS and its use in building distributed applications. A thorough reading will equip a programmer with the necessary information to write simple distributed applications using the ELROS language. For a more comprehensive description of ELROS, including its advanced features, consult the ELROS user's manual [in preparation].

II. Writing Distributed Applications

A typical stand-alone application is composed of program units that include its main program, procedures written by the application programmer and procedures written by other programmers. These units are normally coalesced into one or more libraries and brought together during the linking process, resulting in an executable that runs on a single computer.

A distributed application is also composed of many parts, but differs from a stand-alone application in that these parts are not linked into a single executable. Instead, several executables are produced, each of which normally runs on a different system. Distributed applications are composed from two or more executables, sometimes written by the same programmer and sometimes written by different authors. We call the executables that form a distributed application its *components*.

Distributed application components normally interact with each other in pairs. During a particular interaction, one component requests service from another. We say that the component seeking service acts in the role of a *client*, while the component providing the remote service acts in the role of a *server* (fig. 1).

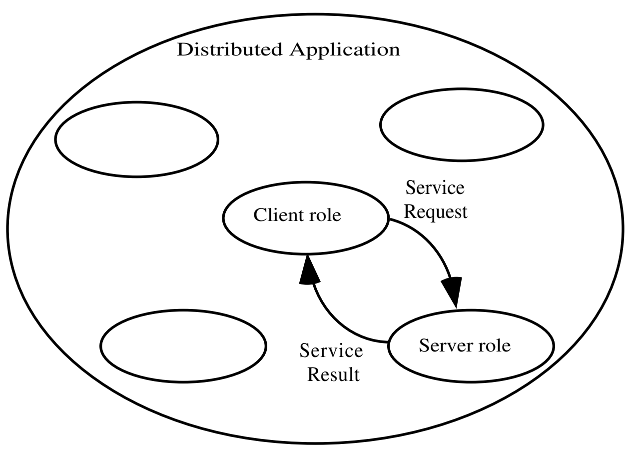

Figure 1 - Distributed application components acting in the role of client and server

In general components are not limited to one role. They may act as a client when interacting with one component and act as a server while interacting with another. It is even common for components to act in the role of a client and concurrently act in the role of a server for another interaction (fig. 2).

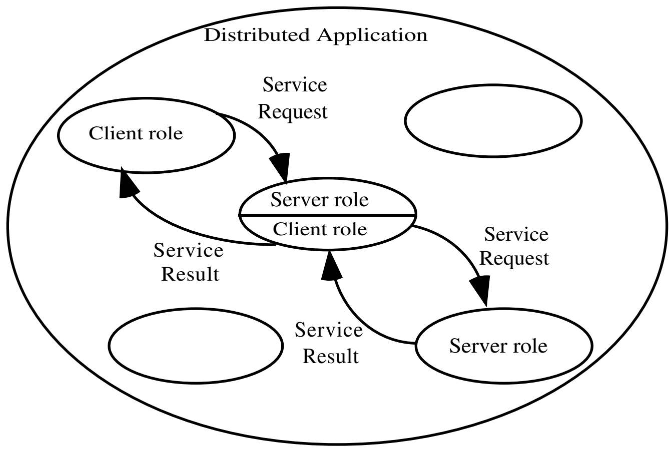

Figure 2 - Distributed application component concurrently acting as both client and server

The actions that a component takes when acting in the role of a client are different from the actions required when it acts in the role of a server. As a client[2], a component must locate the server that provides the remote service, make a service request of it and then accept the results (if any) that it returns. A server waits for a service request, carries out the activity that satisfies the request and then returns any results of that activity to the client.

Clients and servers can interact in two different ways. If the client makes a service request and then blocks its activity until the server returns the results of the request, we say the interaction is *synchronous*. If the client makes a request and then continues computing while the request is being processed by the server, waiting at some later point in time for the results of the request, we say the interaction is *asynchronous*. Generally,

[2]For the sake of simplicity, we shorten the phrase "acting in the role of a client" and "acting in the role of a server" to "client" and "server" respectively. The reader must keep in mind, however, that components can act in either role and consequently can be described as a server and a client with respect to different interactions.

synchronous interactions are useful when the client cannot continue computing until the results of a service request are received. On the other hand, asynchronous interactions are useful in two cases. There are situations in which a client wishes to compute in parallel with the server. This requires it to make asynchronous service requests so that it does not block. Secondly, sometimes a client desires to make several requests of a server at the same time, not blocking on each individual request. This is known as *batching*.

A service request made by a client will generally contain several parameters that specify data required by the server to satisfy the request. Obviously, the client and server must agree on the identity and data types of these parameters so that their interactions are not confusing. Definition of the services offered by a server (and used by a client), including the definition of parameters that each service request takes, is collected in one place called an *interface*. There is a set of ELROS statements used to define interfaces, which are text files input to the ELROS preprocessor. When the ELROS preprocessor compiles a program implementing client and/or server interactions, the program must specify the interfaces it uses. We say a program *imports* the interfaces it needs to carry out its service interactions. There are several ELROS statements that are concerned with defining and using service interfaces.

Finally, interactions between a client and server can generate various errors. For example, if the communications network that connects the machines of the client and server fails, it may not be possible for a client to make a service request or for a server to return the results of a request. If a client mistakenly uses an interface definition different from the one used by the server, various errors are possible, such as service requests unsupported by the server, results incomprehensible to the client, and parameters of the wrong identity or type. ELROS supports an exception facility that allows the programmer to specify what actions to take when such errors occur.

III. Programming Simple Clients And Servers With ELROS

As previously described, a distributed application component acts as either a client or server when interacting with other components. In this chapter we describe the ELROS statements used to program client and server interactions.

3.1 Specifying an interface

There must be agreement between a client and server on the identity and parameters of service requests. A collection of related service request descriptions comprises an interface. ELROS supports the specification of interfaces, which are used by the ELROS preprocessor when clients and servers are compiled.

An ELROS interface has the following general form:

interface *interface-name* ;
  type-definitions
  operation-definitions
end;

The reserved words **interface** and **end** are used to bracket interface definitions. Each interface is given a name (*interface-name*) that identifies it[3]. Interface definition statements are either *type-definitions* or *operation-definitions*, the latter corresponding to the specification of individual service requests[4].

---

[3] The name of each interface imported into a client or server must be unique.

[4] At present ELROS interfaces cannot be nested (we intend to correct this problem in a future version). However, there is a way to achieve the same effect. If ELROS *type-definitions* and *operation-definitions* are placed in a file and the C preprocessor #include command is used within the body of an interface specification to include these definitions, the same *type-definitions* and *operation-definitions* can be used in multiple interfaces.

3.2 Defining data types

Type definitions allow an interface designer to describe the data structures used by operations. Each type definition in the type definitions section is terminated by a semicolon. There are 17 basic data types and four type constructors, the latter being used to create Arrays, Records and Unions. The basic data types are :

Boolean	- a value either **true** or **false**
Integer	- a signed integer value
Real	- a signed floating point value
BitString	- a value consisting of an array of bits
OctetString	- a value consisting of an array of octets
Enumerated	- a discrete set of values defined by the programmer
ObjectIdentifier	- a value consisting of an array of non-negative integers
Null	- no value
Any	- any value[5]
NumericString	- a character string value of numeric characters
PrintableString	- a character string value of printable characters
TeletexString	- a character string value of teletex characters
VideotexString	- a character string value of video characters
VisibleString	- a character string value of visible characters
IA5String	- a character string value of characters from the IA5 set
GraphicString	- a character string value of graphic characters
GeneralString	- a character string value of general characters

The last 8 data types are string types limited to characters from various standard alphabets. However, ELROS does not check to ensure that variables of these types contain only characters from the specified alphabets.

With two exceptions, the specification of a basic ELROS type requires not only identifying the base type (e.g., **Integer**, **BitString**), but also specifying the range of values supported by the type (for the moment we ignore an

---

5 The Any type is useful mostly for programs implementing ISO standard protocols. The use of Any variables is beyond the scope of this primer.

alternative form for “string types” described below). This is accomplished by concatenating to the base type identifier a positive integer (or in some cases a pair of positive integers), contained within parentheses. In most cases the integer specifies the number of octets used to represent the type. For example, an integer type capable of containing values in the range $(-2^{31}, 2^{31}-1)$ would be defined as Integer(4). This follows because in twos-complement arithmetic[6], it takes 32 bits (i.e., 4 octets) to represent integers in that range. Similarly, an OctetString type capable of holding 16 octets would be defined as OctetString(16). The meaning of the positive integer(s) attached to the basic type names are :

Boolean(n)	**n** is the number of octets used to represent the boolean value (all bytes equal to zero representing **false**, all other values representing **true**).
Integer(n)	**n** is the number of octets used to represent the twos-complement integer.
Real(m, e)	**m** is the number of bits used to represent the mantissa and **e** is the number of bits used to represent the exponent. ELROS interprets these numbers as specifying a base 2 normalized format in which the exponent is biased half-way between its range.
BitString(b)	**b** is the number of bits in the BitString.
OctetString(n)	**n** is the number of octets in the OctetString.
Enumerated(n)	**n** is the number of octets used to represent the discrete values.
ObjectIdentifier(c, o)	**c** is the maximum number of components in the ObjectIdentifier and **o** is the number of octets per component.
Null	Null represents a type with no value, so no integer is specified.
Any	Any represents an indeterminate type, so no integer is specified.

[6] Signed integers in ELROS are assumed to be representated in twos-complement arithmetic.

<chartype>String(n)	**n** is the number of characters in the character string.

The ELROS preprocessor translates each of these types into C typedef equivalents. This translation is controlled by the *parameters.dat* file read by the preprocessor each time it runs. For a typical machine the ELROS types might be translated according to the following rules (NB: ceiling(*expression*) represents the smallest integer greater than or equal to *expression*):

Boolean(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Boolean(1) -> Boolean(2) translates to **typedef unsigned short**, Boolean(3) -> Boolean(4) translates to **typedef unsigned int**, Boolean(5) -> Boolean(8) translates to **typedef unsigned long**, and Boolean(n), where n > 8, translates to **typedef struct { unsigned char val[n] ;}**.
Integer(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Then Integer(1) -> Integer(2) translates to **typedef short**, Integer(3) -> Integer(4) translates to **typedef int**, Integer(5) -> Integer(8) translates to **typedef long**, and Integer(n), where n > 8, translates to **typedef struct { unsigned char val[n]; }**.

Real(m, e)	Assume **float** represents a number with 24 mantissa bits and 8 exponent bits, and **double** represents a number with 53 mantissa bits and 11 exponent bits. Then Real(m,e), where $m \leq 24$ and $e \leq 8$, translates to **typedef float**; Real(m,e), where either $24 < m \leq 53$ and $e \leq 11$ or $m \leq 53$ and $8 < e \leq 11$, translates to **typedef double**; and Real(m,e), where either $m > 53$ or $e > 11$ translates to **typedef struct { unsigned char val[x]; }**, where x = ceiling(m/8) + ceiling(e/8) + 1.
BitString(b)	translates to **typedef struct { unsigned long len; unsigned char val[n] ;}**, where n = ceiling(b/8).
OctetString(n)	translates to **typedef struct { unsigned long len; unsigned char val[n] ;}**.
Enumerated(n)	Assume a **short** is represented as two bytes, an **int** as four bytes, and a **long** as eight bytes. Then Enumerated(1) -> Enumerated(2) translates to **typedef short**, Enumerated(3) -> Enumerated(4) translates to **typedef int**, Enumerated(5) -> Enumerated(8) translates to **typedef long**, and Enumerated(n), where n > 8, translates to **typedef unsigned char [n]**.
ObjectIdentifier(c, o)	translates to **typedef struct { int size, unsigned long comp[c];}**. **size** represents the number of components in the ObjectIdentifier. **o** must be less than or equal to sizeof(unsigned long).
Null	translates to **typedef void ***.
Any	translates to **typedef struct { unsigned char *pointer; unsigned long count; unsigned long size;}**. *pointer* holds the address of the buffer containing the Any value, *count* is the number of bytes in that buffer used to represent the value, *size* is the length of the buffer, and *sig* is used for internal checking.

<NULTerm>String(n)	translates to **typedef char [n+1]**, where NULTerm is either Numeric, Printable, Visible, or Graphic. Strings of zero length are represented by a single NUL character.
<LenTerm>String(n)	translates to **typedef struct { unsigned long len; char val[n];}** where LenTerm is either Teletex, Vidoetex, IA5, or General.

The distinction between NULTerm and LenTerm character string types is that NULTerm types are terminated by the NUL character, while LenTerm types, which may validly contain NUL characters, are terminated by the value in the "len" field.

There is an optional representation for certain ELROS basic types, called the *string types*, which consist of **BitString**, **OctetString**, **ObjectIdentifier** and the 8 character string types. This representation substitutes an indefinite value (represented by the character "*") in place of the string length or number of ObjectIdentifier components. The ELROS preprocessor translates these indefinite length representations into the following C typedef equivalents.

BitString(*)	translates to **typedef struct { unsigned long len; unsigned char *val; }**, where **len** = ceiling(b/8), b is the number of bits, and **val** holds the address of the BitString value.
OctetString(*)	translates to **typedef struct { unsigned long len; unsigned char *val; }**, where **val** holds the address of the OctetString value.
ObjectIdentifier(*, o)	translates to **typedef struct { int size, unsigned long *comp;}**. **size** represents the number of components in the ObjectIdentifier. **o** must be less than or equal to sizeof(unsigned long) and **comp** holds the address of the ObjectIdentifier value.

<NULTerm>String(n) translates to **typedef char ***, where NULTerm is either Numeric, Printable, Visible, or Graphic. Strings of zero length are represented by a pointer to a single NUL character, not by a NULL pointer.

<LenTerm>String(n) translates to **typedef struct { unsigned long len; char *val;}** where LenTerm is either TeletexString, VidoetexString, IA5String, or GeneralString.

To use variables of these types, the programmer must ensure that the relevant variable pointer references memory holding the variable value.

An example illustrates the use of the basic types in an interface specification. Suppose we are writing an interface for a set of client/server interactions to compute the properties of a gas located somewhere between the center of a star and 10 star diameters distant. We parameterize a computation to compute the density of a gas by specifying its pressure and temperature at a particular position. The client provides an integer value that identifies the gas as well as the values of the pressure and temperature. The server computes the density for these parameters and returns it to the client.

Four types are required. Three reals are used for the pressure, temperature and density and an integer is used to identify the gas. The pressure ranges from 10^{-50} to 10^{50} atmospheres and requires 12 bits of precision, temperature ranges from 1° to 10^{10}° Kelvin and requires 14 bits of precision, and density ranges from 10^{-10} to 10^{50} particles/cm^3 and requires 48 bits of precision. There are 10 gases to be identified. The following types are used[7].

---

[7] The exponents in the definition of the Real variables are computed according to the following argument. The pressure values have exponents with a range of 10^{100}, which represents a range of 2^{333}. Representing an exponent range of 333 requires 9 bits. The temperature values have exponents with a range of 10^{10}, which represents a range of 2^{34}. However, since the exponent is biased half-way between its range, we must use 10^{20} when calculating the ELROS exponent value (i.e., 10^{-10} to 10^{10}), implying a range of 2^{66}. Representing an exponent range of 66 requires 7 bits. The density values have

Integer(1) – for the gas argument
Real(12,9) – for the pressure argument
Real(14,7) – for the temperature argument
Real(48,9) – for the density argument

This example illustrates an important principle of ELROS interface design. It is fairly probable that some client of a service interface will run on a machine with data representations different than the server machine (e.g., the client runs on a SUN with IEEE floating point formats, while the server runs on a CRAY with its own proprietary floating point format). ELROS correctly handles the data format conversions that are required in this situation. However, if the interface designer specifies a range of values based on a particular architecture, for example choosing values for the mantissa and exponent ranges that are valid for a CRAY, and these ranges cannot be represented on the client's machine architecture (e.g., a SUN), ELROS maps the client's declaration into an array of unsigned characters.

For this reason it is extremely important that interface designers declare basic type ranges based on the requirements of the client and server algorithms and **not** on the basis of any particular architecture. ELROS will handle the mapping of the ELROS data type into the appropriate native data type on the target machine.

3.3 Defining synchronous operations

Operations are the second class of interface objects defined in an interface specification. In order to support both synchronous and asynchronous service requests, ELROS supports two kinds of operation. From the client's perspective, synchronous operations act very much like normal procedure calls, i.e., the client blocks within the operation until the server returns the operation's results. Synchronous operations are defined using the reserved word **call**. Asynchronous operations do not cause the

---

exponents that can be as great as 10^{50}. This implies an ELROS exponent range of 10^{100}, which represents a range of 2^{333}. Representing an exponent range of 333 requires 9 bits.

client to block when they are called. Asynchronous operations are defined using the reserved word **invoke**.

ELROS operations (both synchronous and asynchronous) come in two varieties: 1) those taking positional parameters, and 2) those taking named parameters (fig. 3).

Operation Definitions	Synchronous	Asynchronous
Positional Parameters	**call** *op* $p_1, p_2,..., p_n$ (p_i = basic or defined type)	**invoke** *op* $p_1, p_2,..., p_n$ (p_i = basic or defined type)
Named Parameters	**call** *op* $n_1, n_2,..., n_n$ (n_i = named parameter)	**invoke** *op* $n_1, n_2,..., n_n$ (n_i = named parameter)

Figure 3 - The four kinds of service requests

We concentrate first on synchronous operations taking positional parameters.

When defining a synchronous operation, we must indicate the name of the operation, its input parameters, its output parameters and those parameters which are used both for input and output. For example, we might name an operation to compute the density of a gas at a given pressure and temperature – "compute_density." We would define this operation with the following statement in the interface specification :

```
call        compute_density      10
            in Integer(1), Real(12,9), Real(14,7),
            out Real(48,9) ;
```

This definition specifies that the operation compute_density takes four parameters, three input parameters and one output parameter. The input parameters correspond to the identifier of a gas – Integer(1), the pressure of the gas – Real(12,9), and the temperature of the gas – Real(14,7). That these are input rather than output or input/output parameters is indicated by the reserved word **in** that precedes them. Notice that more than one input parameter can follow the **in** keyword and that all parameters are separated from each other by commas. The output parameter corresponds to the density of the gas - Real(48,9) and is specified as an output parameter by following the reserved word **out**[8].

The identifier of the operation, "compute_density," is followed by the integer 10, which is its operation number. All operations, whether they are synchronous or asynchronous, require an integer operation number[9] that is unique within each operation class[10] in the interface specification. Note that the reserved word **call**, the operation identifier, the operation number and the parameter list are separated by white space, while the parameters within the parameter list are separated by commas.

While this definition of compute_density is technically correct, it is a bit clumsy to read because the arguments are non-descriptive. This can be rectified by defining descriptive types for the arguments and using these in the compute_density definition. For example we could make the following type definitions:

```
Integer(1)     gas_identifier;
Real(12,9)     pressure_value;
Real(14,7)     temperature_value;
Real(48,9)     density_value;
```

Using these definitions we can define compute_density as follows :

```
call     compute_density     10
         in gas_identifier, pressure_value, temperature_value,
         out density_value;
```

---

[8]While not illustrated by this example, a call operation may also specify inout parameters, these being preceded by the reserved word **inout**.

[9] Actually, operations numbers also may be ObjectIdentifiers. However, this is an advanced topic outside the scope of the primer.

[10] Operations are grouped into four classes : 1) invokes, 2) results, 3) errors and 4) rejects. The first three classes are discussed below. Call operation numbers must be unique with respect to both invoke and result operations. Reject operations are not covered in the primer.

Assuming that we use the identifier "physics" to name the interface that contains the compute_density operation, the complete interface specification would look like :

```
interface physics;

    Integer(1)     gas_identifier;
    Real(12,9)     pressure_value;
    Real(14,7)     temperature_value;
    Real(48,9)     density_value;

    call      compute_density     10
              in gas_identifier, pressure_arg, temperature_arg,
              out density_arg;

end;
```

3.4 Using synchronous operations

Once we have defined an interface, we are in a position to write a client and server that uses its operations. There are two activities that must be programmed for client-server interactions: 1) making contact between the client and server, and 2) making and handling service requests between the client and server.

The requirement to make contact between a client and server may at first seem a bit strange to someone only familiar with programming on stand-alone computer systems. This is because contact between a calling program and the procedure it calls is normally carried out when the executable is linked. However, in a distributed application, the client and server normally run on different machines, so they must make contact at runtime.

Since distributed application components may be engaged in several interactions concurrently, there must be some way to discriminate between them. In ELROS we use the notion of a communications instance to do this. Communications instances are much like open files and communications instance variables are similar to file descriptors in that they represent the state of an on-going interaction. In particular, a communications instance represents an active connection between the client and server. Programs are allowed to have many communications instances active concurrently and are even allowed to have multiple communications instances to the same server active contemporaneously.

A communications instance variable is declared with the **control** reserved word. For example the following ELROS statement declares two instance variables[11], inst1 and inst2 :

```
control(physics)  inst1, inst2;
```

Notice in the instance variable declaration we have placed the word "physics" in parentheses after the reserved word **control**. This is the name of the interface that includes the definitions of operations and data types that will be used over the instance.

Since the ELROS preprocessor must be able to read and process this interface, we must tell it the name of the file in which the interface is located. This is called **importing** the interface and we do this with an include statement. Suppose the interface "physics" is stored in a file with the pathname "~/dist/interfaces/physics.i". This interface is made known to the ELROS preprocessor by including in the embedded language program the statement :

```
#include  "~/dist/interfaces/physics.i"
```

The include statement must appear before the declaration of any instances that use the physics interface.

The include statement and instance declaration are the first statements we have described that are inserted by the programmer into C programs directly, rather than appearing in interface specifications. Interface specifications usually are located and maintained in files separate from those containing the ELROS embedded language programs in order to share them between different programs (fig. 4).

---

[11] To economize writing style, we often refer to communications instance variables simply as instance variables.

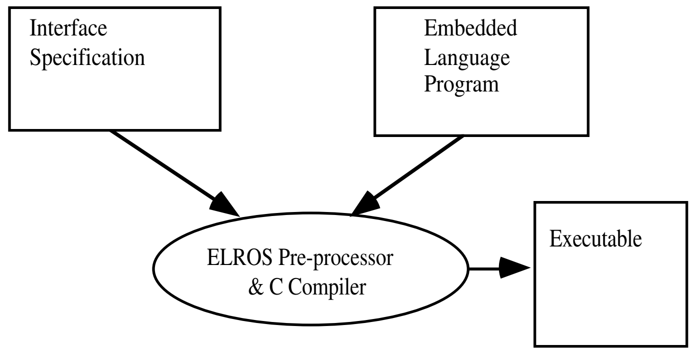

Figure 4 - ELROS preprocessor input and output

3.4.1 Writing the client

Once we have imported the appropriate interface and declared an instance variable associated with it, we are in a position to program the contact between a client and server. In ELROS a client uses the **bind_addr** statement to contact a server. **bind_addr** takes four arguments : 1) an instance variable, 2) rules, 3) a pointer to an address structure, and 4) a timeout value. The instance variable is the output parameter of **bind_addr**, being set to a value that can be used by other ELROS statements (much like a file descripter is initialized by an open call). The rules argument is an **enum** type with two choices, asn_1_ber and asn_1_ler[12]. This argument selects the data encoding rules used by ELROS when transfering data from one machine format to another. The timeout value is used to timeout ELROS statements that block (including **bind_addr** statements) as well as the underlying communication routines in the ELROS runtime system.

The address structure argument contains information that identifies the server. It has the following general format:

```
struct address
    {
    bind_platform platform; /* isode, puresockets */
    bind_role role; /* initiator, responder */
    union
```

[12] The value asn_1_ber selects normal encoding rules, while asn_1_ler selects encoding rules optimized for the transfer of large volumes of scientific data (i.e., reals and integers).

```
        {
        struct
              {…

              }  isode_addr;

        struct
              {…

              }  puresocket_addr;
        } un;
  };
```

This structure is available as a typedef with the identifier **bind_address**. (**NB**: not all elements of the address structure are shown above, only those that are important to the application programmer. Consequently, application programs should always use the **bind_address** typedef when declaring address structures).

The address structure argument must be filled in by the client before calling **bind_addr**. To ease the programming chore this imposes, platform dependent routines are provided that initialize the address structure and fill in the appropriate content. For the puresocket platform this routine is **init_puresocket_addr**(&addr, role, "IP_address+port"). The first argument is a pointer to the address structure. The second argument specifies the role the component is taking in interactions over the commmunications instance[13]. In the case of the client, this role is **initiator**. The third argument is a string consisting of the dotted IP address followed by the "+" character and then the port.

For example, suppose we are writing a client that contacts a server implementing the compute_density operation described above. If the client and server interact using the puresocket communications platform and the server is at IP address 134.8.20.12 and port 2047, the client code to contact

---

[13] The "role" argument is of type "bind_role", which is an enum with constants "initiator" and "responder".

the server would look something like the following (assuming a timeout value of 10 seconds) :

```
#include "~/dist/interfaces/physics.i"

control(physics) physics_server;
bind_address addr;

init_puresocket_addr(&addr, initiator, "134.8.20.12+2047");

bind_addr(physics_server, asn_1_ber, &addr, 10);
```

Once a communications instance is bound by executing **bind_addr**, it can be used by the client to execute a compute_density operation with the ELROS **call** statement, which has the following form :

```
compute_density@physics_server(gas, pressure, temperature, density);
```

The four parameters "gas", "pressure", "temperature", and "density" must have been previously declared and set to appropriate values (except for "density", which is an output parameter). Declaration of these variables is possible using the same ELROS type identifiers that are used in the interface specification for defining the formal arguments to "compute_density"[14]. For example, the following declarations would be typical :

```
gas_identifier        gas;
pressure_value        pressure;
temperature_value     temperature;
density_value         density;
```

[14] They also can be declared with the appropriate C programming language declarators, if that is desirable. However, using the ELROS types for declaring variables in embedded language programs ensures portability. Declaring these variables with the corresponding C types almost guarantees the embedded language programs will not be portable.

After calling the compute_density operation and obtaining its results, the client should let the server know that no more operations will arrive over the communications instance (this is like closing an open file). Consequently, the client should unbind the instance with the ELROS **unbind** primitive before proceeding with any calculations. Unbind takes one argument, a bound instance variable.

Putting all the above pieces together, we get the following client code that will contact the physics server and request it to compute the density of the gas Argon at a pressure of 10^{20} atmospheres and a temperature of 10^6 ° Kelvin (assume the integer identifier of Argon is 18, its atomic number).

```
/* import the physics interface */

    #include  "~/dist/interfaces/physics.i"

/* declare the instance and address structure variables */

    control(physics)  physics_server;
    bind_address  addr;

/* declare the compute_density operation arguments */

    gas_identifier           gas;
    pressure_value           pressure;
    temperature_value        temperature;
    density_value            density;

/* initialize the address structure and fill in the server's address */

    init_puresocket_addr(&addr, initiator, "134.8.20.12+2047");

/* establish communications with the server */

    bind_addr(physics_server, asn_1_ber, &addr, 10);

/* initialize the compute_density arguments */
```

```
    gas = 18;
    pressure = 1e20;
    temperature = 1e6;

/* request the server to compute the density of Argon at 1e20 atmospheres
    and 1e6 degrees Kelvin */

    compute_density@physics_server(gas, pressure, temperature, density);

/* since the client has no more use for the communications instance,
    unbind it */

    unbind(physics_server);

/* use the density in whatever calculations are desired */

    <remainder of program>
```

3.4.2 Writing the server

A component acting in the role of a server waits to be contacted by a client and then carries out the operations the client requests. Servers wait for contact by executing **bind_addr**, specifying the role of **responder**. As with a client, the server must first initialize an address structure. For example, the following code would be typical for the physics_server described above :

```
/* import the physics interface */

    #include "~/dist/interfaces/physics.i"

/* declare the instance and address structure variables */

    control(physics) physics_client;
    bind_address addr;
```

```
/* initialize the address structure and fill in the server's address */

     init_puresocket_addr(&addr, responder, "134.8.20.12+2047");

/* establish communications with the client */

     bind_addr(physics_client, asn_1_ber, &addr, 10);
```

Note that this code is almost identical to that executed by the client, except init_puresocket_addr is called with its role parameter set to "responder". Since the role of the address structure specifies responder, **bind_addr** will block waiting for service requests to arrive at this address, rather than trying to contact a component at the specified address.

Once contact is made with a client, the server must accept service requests, carry out the activity to satisfy them and return any results of the requests to the client. Normally, a server supports more than one operation; although, in our simple example this is not so. Consequently, the server must first determine which of its operations the client is invoking. ELROS provides the **accept** statement for this purpose. It begins with the reserved word **accept**, followed by an instance variable (which must represent an active instance bound by a previous call to **bind_addr**) and then a series of on-clauses, each of which specify a particular operation in the service interface. The format for the **accept** statement is :

accept *instance-variable*
    {
    on *operation*$_1$*(parameter-list)*
        C-statement$_1$
    on *operation*$_2$*(parameter-list)*
        C-statement$_2$
            .
            .
    on *operation*$_n$ *(parameter-list)*
        C-statement$_n$
    }

The **accept** statement is tied to a particular communications instance by the instance variable given immediately following the **accept** reserved word. Following the instance variable and contained within opening and closing curly braces is a series of on-clauses, each of which specifies an operation in the interface to which the instance variable was associated when it was declared. The operation specified in each on-clause takes a parameter list, which specifies variables into which the ELROS runtime system stores the values communicated by the client. The *C-statement* following each operation specification can be any general C language statement, typically a compound statement. It may contain not only C programming constructs, but also other ELROS embedded language statements, including declarations, **bind_addr** statements, **call** statements, **accept** statements and **give** statements (covered below).

When the **accept** statement is executed, the program blocks waiting for an operation to arrive over the instance. When an operation is received, control passes to the appropriate on-clause. Control also passes out of the **accept** statement if an operation doesn't arrive within the timeout period specified in the **bind_addr** call. This error condition can be detected using the **exceptions** clause described in section 4.4.

For example, the code necessary to implement the physics server discussed above might look like :

```
/* import the physics interface */

    #include  "~/dist/interfaces/physics.i"

/* declare the instance and address structure variables */

    control(physics)  physics_client;
    bind_address  addr;

/* declare the compute_density operation arguments */

    gas_identifier          gas;
    pressure_value          pressure;
```

```
    temperature_value       temperature;
    density_value           density;

/* initialize the address structure and fill in the server's address */

    init_puresocket_addr(&addr, responder, "134.8.20.12+2047");

/*  establish communications with the client */

    bind_addr(physics_client, asn_1_ber, &addr, 10);

/* obtain a compute_density request and service it */

    accept physics_client
        {
        on compute_density(gas, pressure, temperature)
              {
              density = density_calculation(gas, pressure, temperature);

              give physics_client
                    {
                    compute_density(density);
                    }
              }
        }
```

There are two notable things about this code fragment. First, only the **in** parameters are specified in the parameter list for the compute_density on-clause. This is so because these are the only parameter values sent by the client when it executes the compute_density **call** statement.

Secondly, there is a new ELROS statement introduced in the code fragment – the **give** statement. The **give** statement is used by servers to return the results of an operation to the client.

The format of the **give** statement is :

give *instance-variable C-statement*

The **give** statement begins with the reserved word **give**, followed by an instance variable and then by a C language statement (which may be a compound statement and include other ELROS embedded language statements). To send the results of an operation to the client, the programmer embeds the name of the operation in the C statement, giving within parentheses an expression for each of its **out** parameters and separating these with commas. In the example shown above, the variable "density" is the argument for the single compute_density out parameter.

Give statements are the means by which results are returned by a server to its client. However, in section 4.6 we will see that **give** statements can be used in more general ways to support asynchronous operations. We defer until that section a description of the full generality and power of **give** statements.

3.5 Exceptions

The client and server developed in the previous sections have a significant deficiency. The client contacts the server, requests the density of a gas at a particular pressure and temperature, unbinds the communications instance and continues computing. The server waits for a compute_density request to arrive, computes it, returns the results and then terminates. This strategy is not going to work well if the density of more than one gas at a particular pressure and temperature is required by the client. The strategy is also deficient if the server interacts with more than one client.

The obvious solution to this problem is to program the client so it executes calls to the server in a loop. The server can also be programmed to wait for contact from a client and then loop servicing new compute_density requests. The client code fragment that does this looks like :

```
/* declare and initialize variables */

     <declarations and initializations>

/* establish communications with the server */
```

```
    bind_addr(physics_server, asn_1_ber, &addr, 10);

/* request the server to compute the density of several gases at 1e20
    atmospheres and 1e6 degrees Kelvin */

    gas_identifier  gases[8] = {1, 2, 7, 8, 9, 10, 17, 18};
    density_value densities[8];

    for(i=0; i<8; i++)
        {
        compute_density@physics_server(gases[i], pressure, temperature,
                                                    densities[i]);
        }

/* since the client has no more use for the communications instance,
    unbind it */

    unbind(physics_server);

/* use the densities in whatever calculations are desired */

    <remainder of program>
```

Writing the server code fragment is a little more problematical, since, as yet, we have not described a way for the server to know when the client is finished with the communications instance (i.e., when the client executes an **unbind**). ELROS provides a mechanism for the server to discover this, which is also used by both clients and servers to react to errors. We concentrate here on how a server is informed when a client unbinds an instance.

ELROS allows the attachment of an **exceptions** clause to many of its executable statements, including the **accept** statement. The **exceptions** clause begins with the reserved word **exceptions** and is followed by a set of on-clauses (much like the structure of an **accept** statement) that specify certain exceptional termination conditions, each followed by statements to execute when those conditions occur. The "done" condition represents an **unbind** by a correspondent (this condition is specified by the reserved word **on** followed by the reserved word **done**).

When the server executes an **accept** statement with an attached **exceptions** clause containing a **done** on-clause and the client executes an **unbind**, control in the server passes to the code immediately after the **on done** reserved words. For example, suppose we attached the following **exceptions** clause to the **accept** statement in the compute_physics server :

```
accept physics_client
    {
    on compute_density(gas, pressure, temperature)
          {
          density = density_calculation(gas, pressure, temperature);

          give physics_client
                {
                compute_density(density);
                }
          }
    }
exceptions
    {
    on done
          unbind(physics_client);
    }
```

When the client unbinds the communications instance, control in the server passes from the **accept** statement to the **done** on-clause within the **exceptions** clause, causing the server to unbind its instance variable.

This still doesn't allow the server to handle more than one compute_density request in a single execution. The **accept** statement is not a looping statement. It executes once, either branching to the compute_density on-clause, to the **done** on-clause or timing out. To accept more than one compute_density request, the **accept** statement must be placed in either a **for** or a **while** loop. Using a **while** loop, the complete server code fragment is :

```
#define FALSE 0
#define TRUE 1

    short bound;

/* import the physics interface */

#include  "~/dist/interfaces/physics.i"

/* declare the instance and address structure variables */

    control(physics)  physics_client;
    bind_address  addr;

/* declare the compute_density operation arguments */

    gas_identifier          gas;
    pressure_value          pressure;
    temperature_value       temperature;
    density_value           density;

/* initialize the address structure and fill in the server's address */

    init_puresocket_addr(&addr, responder, "134.8.20.12+2047");

/*  establish communications with the client */

    bind_addr(physics_client, asn_1_ber, &addr, 10);

/* obtain the compute_density requests and service them. When the client
    unbinds, unbind the communications instance and exit the while loop */
```

```
bound = TRUE;

while(bound)
    {
    accept physics_client
          {
          on compute_density(gas, pressure, temperature)
                {
                density = density_calculation(gas, pressure, temperature);

                give physics_client
                      {
                      compute_density(density);
                      }
                }
          }
    exceptions
          {
          on done
                {
                unbind(physics_client);
                bound = FALSE;
                }
          }
    }
```

The details of the **while** loop may bother some programmers concerned with good programming style. They might suggest using a **break** statement within the **done** on-clause to exit the **while** loop. This would be a grave error. The ELROS preprocessor translates **accept** statements and exceptions clauses into **switch** statements. Consequently, the use of **break** within these statements is dangerous. **Break** may only be used to re-route control within a C language statement embedded in an ELROS statement (e.g., within a **while** loop embedded in the C language statement part of an on-clause). "Break" must never be used to re-route control outside of an **accept** statement or on-clause.

IV. Using Some Advanced Features of ELROS

So far we have concentrated on how to program a simple client and server using ELROS. While the ELROS constructs we have discussed up to this point are useful in most situations, many distributed applications either require or would benefit from the use of some of the more advanced features of ELROS. In this chapter, we cover some of these. We have mentioned previously the availability in ELROS of type constructors for Arrays, Records and Unions. We now describe these in detail.

4.1 Arrays

4.1.1 Simple Arrays

The ELROS type constructors allow the programmer to define types in an interface specification that can then be used as arguments in a remote operation within an embedded language program. Arrays of data elements are defined using the Array type constructor :

Array *element_type name_of_array_type* [*number_of_elements*]

Element_type may be any type, including one of the basic types as well as a type defined previously in the interface specification. The programmer identifies the Array type with *name_of_array_type*, which follows the *element_type*. The *number_of_elements* in the Array is given in square brackets following *element_type* and *name_of_array_type*.

For example, suppose instead of computing the density of a gas at a single pressure and temperature, we wish to calculate the density of a gas at 20 pressure-temperature pairs. We accomplish this by defining the pressure, temperature and density parameters to compute_density to be Arrays, rather than single values. The physics interface specification would then look like :

```
interface physics;

    Integer(1) gas_identifier;
    Real(12,9) pressure_value;
```

```
    Real(14,7) temperature_value;
    Real(48,9) density_value;

    Array pressure_value pressure_array[20];
    Array temperature_value temperature_array[20];
    Array density_value density_array[20];

    call    compute_density    10
                in gas_identifier, pressure_array, temperature_array,
                out density_array ;

end;
```

Once pressure_array, temperature_array and density_array are defined as types in the interface specification, they are available for use in embedded language programs[15]. To use the new compute_density service, the client and server would be modified as follows :

<u>Client</u>

```
/* import the physics interface and declare the instance and address
   structure variables */
                    .
                    .
                    .
/* declare the compute_density operation arguments */

   gas_identifier       gas;
   pressure_array       pressure;
   temperature_array    temperature;
   density_array        density;
```

---

[15] Note, however, that the type constructors themselves may be used only in interface specifications. A declaration such as "array Real(2,2) pressure_array[20]" is not a valid ELROS embedded language statement.

```
/* initialize the address structure and fill in the server's address, establish
     communications with the server */
                                        .
                                        .
                                        .
/* initialize the compute_density arguments */

     gas = 18;
     pressure[0] = 1e20;
     temperature[0] = 1e6;
     pressure[1] = 1e19;
     temperature[1] = 9e5;
          .
          .
          .
/* request the server to compute the density of Argon at the given pressure
     and temperature pairs */

     compute_density@physics_server(gas, pressure, temperature, density);

/* unbind the instance and use the densities in whatever calculations are
     desired */

     <remainder of program>
```

<u>Server</u>

```
/* define FALSE, TRUE and the control variable "bound", import the
     physics interface and declare the instance and address structure
     variables */
                                        .
                                        .
                                        .
/* declare the compute_density operation arguments */

     gas_identifier          gas;
```

```
    pressure_array          pressure;
    temperature_array       temperature;
    density_array           density;

/* initialize the address structure and fill in the server's address and
    establish communications with the client */
                        .
                        .
                        .
/* obtain the compute_density requests and service them. When the client
    unbinds, unbind the communications instance and exit the while loop */

    while(bound)
        {
        accept physics_client
            {
            on compute_density(gas, pressure, temperature)
                {
                density_calculation(gas, pressure, temperature, density);

                give physics_client
                    {
                    compute_density(density);
                    }
                }
            }
        exceptions
            {
            on done
                {
                unbind(physics_client);
                bound = FALSE;
                }
            }
        }
```

Although the **accept** statement in the new compute_physics server looks very much like the statement in the old server, the service is different. The new server accepts an Array of 20 pressure-temperature pairs and calls the routine density_calculation with Array arguments, rather than scalar arguments. This requires the addition of a density Array argument to the density_calculation routine, since the result can no longer be returned to the caller using an assignment statement.

4.1.2 Specifying Array lengths at runtime

One potential problem with the above solution is it requires the client to always send 20 pressure-temperature pairs. Consequently, if the client wishes to compute the density of a gas at only 10 pressure-temperature pairs, it cannot use the compute_physics server shown above.[16]

ELROS provides a feature that allows the establishment of Array lengths at runtime. To use this feature, Arrays are defined as before, except the number of elements given in the interface definitions represents an upper bound of the valid elements in the Array, rather than the actual number. To specify the number of valid elements when an operation is called, an integer expression contained within braces is appended to the Array argument. The value of the expression specifies the number of elements to send. For example, suppose the client wishes to compute the densities of only 10 pressure-temperature pairs located in the first 10 elements of the pressure and temperature Arrays. The following call to compute_density accomplishes this :

```
/* request the server to compute the density of Argon at 10 given pressure
    and temperature pairs */

    compute_density@physics_server(gas, pressure{10}, temperature{10},
                                                       density);
```

[16] In particular, when density_calculation is called, the last ten elements of pressure and temperature will have garbage in them. This could very possibly cause a floating point overflow or underflow, which would cause a server error.

Since "density" is an output parameter, there is no need to specify how many elements it will receive, we know it will be 10 because we know how the server works.

Sometimes we do not know how many elements of an Array a server will return. ELROS also provides for this situation. For output parameters we may place an integer variable in braces[17] to receive the number of elements returned by the server. The value of the variable before the operation call must equal the maximum number of elements that the Array can hold. When the operation completes, the variable will equal the number of elements returned in the Array. The following illustrates this mechanism using the compute_density call.

```
/* request the server to compute the density of Argon at 10 given pressure
    and temperature pairs and receive an indication of how many densities
    were calculated */

    int n = 20;
                        .
                        .
                        .
    compute_density@physics_server(gas, pressure{10}, temperature{10},
                                         density{n});

    /* at this point n should equal 10 */
```

4.1.3 Using Arrays allocated from the heap

In the preceding examples, space for the pressure, temperature and density Arrays is allocated from the stack when the program unit containing the code fragment is called. Sometimes it is desirable to allocate space for these Arrays from the heap, e.g., when the default number of elements in an Array given in the interface specification is large, but the

[17] Actually, the programmer can place any expression inside the braces that evaluates to an integer lvalue.

typical number isn't. To do this we need to declare an Array pointer, which will receive the address of an Array allocated from the heap. Such a declaration is made in the normal C programming language style :

```
pressure_value          *pressure;
temperature_value       *temperature;
density_value           *density;
```

Space is then allocated from the heap using malloc :

```
pressure = (pressure_array *) malloc(10*sizeof(pressure_value));
temperature = (temperature_array *) malloc(10*sizeof(temperature_value));
density = (density_array *) malloc(10*sizeof(density_value));
```

Since specifying the name of a Real Array in a procedure call is equivalent to specifying a Real pointer variable, the use of space allocated from the heap does not change the call to compute_density. However, the density_array is only 10 elements long in this example, so the maximum length of the Array is no longer the default length that appears in the interface specification. Consequently, the variable **n** is set to 10 rather than 20. The resulting call is :

```
int n = 10;
                    .
                    .
                    .
compute_density@physics_server(gas, pressure{10}, temperature{10},
                                    density{n});
```

4.1.4 Receiving Arrays at the server

To this point we have described how a client uses Arrays, the length of which are specified at runtime. Servers use the same ELROS constructs to receive and send Array arguments. For example, the following modifications to the physics_server **accept** statement allows it to receive Arrays with lengths specified at runtime :

```
int pressure_count ;
int temperature_count;
                    .
                    .
                    .
while(bound)
    {
    pressure_count = 20;
    temperature_count = 20;

    accept physics_client
          {
          on compute_density(gas, pressure{pressure_count},
                              temperature{temperature_count})
                {
                if( pressure_count != temperature_count)
                      {
                      give physics_client
                            {
                            compute_density(density{0});
                            }
                      }
                else
                      {
                      density_calculation(gas, pressure_count, pressure,
                                                temperature, density);

                      give physics_client
                             {
                             compute_density(density{pressure_count});
                             }
                      } /* end of if-else */
                } /* end of on compute_density */
```

```
        } /* end of accept */
    } /* end of while */
```

In this **accept** statement the lengths of the arguments "pressure" and "temperature" are placed by ELROS respectively in "pressure_count" and "temperature_count" when the **accept** statement executes ("pressure_count" and "temperature_count" initially contain the maximum length of the Array so that the ELROS generated code knows how much space is available for storing the values received from the correspondent).

If "pressure_count" and "temperature_count" are unequal, the server returns an error indication by executing a **give** statement in which the length of the argument "density" is zero[18]. Otherwise, a call is made to the procedure density_calculation, which now takes an extra parameter specifying how many densities to compute, and the results of the calculation are returned to the client with the subsequent **give** statement. The value contained in the braces associated with the parameter to the operation within the **give** statement specifies how many elements in the Array are valid and should be transmitted.

4.1.5 DynamicArrays

There is another array constructor available to the ELROS programmer, which is useful when a programmer wishes to decide at runtime the number of array elements to send to a correspondent and the array is embedded within another data structure. Arrays of this kind are defined using the DynamicArray type constructor :

DynamicArray *element_type name_of_array_type*

Element_type may be any type, including one of the basic types as well as a type defined previously in the interface specification. The programmer

[18] This is not the ideal way to convey error conditions. ELROS has an application error reporting mechanism that we discuss in section 4.7.

identifies the DynamicArray type with *name_of_array_type*, which follows the *element_type*.

This ELROS type constructor translates to the following typdef :

```
typedef struct
        {
        unsigned long len;
        element_type  *val;
        } name_of_array_type;
```

The len field specifies the number of elements in the DynamicArray and the val field contains a pointer to the DynamicArray data. For example, suppose the following type declaration appears in an interface specification:

```
DynamicArray  pressure_value  pressure_array;
```

This defines a type with the name pressure_array. Suppose the following declaration appears in an embedded language program:

```
pressure_array  pressure;
```

The ELROS preprocessor translates it into the following C declaration:

```
struct
   {
   unsigned long len;
   pressure_value  *val;
   }  pressure;
```

A DynamicArray is treated differently by the ELROS runtime system depending on whether the value it contains is being sent, i.e., used in a **call** or **give** statement, or received, i.e., used in an **accept** statement. When being sent, the runtime system assumes the programmer has stored the appropriate values in "len" and "val". That is, the runtime system treats the memory pointed to by "val" as a buffer containing "len" elements of *element_type*. In the example, the runtime system assumes “val” points to a buffer containing “len” pressure_value elements.

When the runtime system is receiving into a DynamicArray, on the other hand, it allocates enough space from the heap (using malloc) to hold the elements sent. Consequently, any pointer value in “val” is clobbered on a receive. The “len” field of the DynamicArray struct is set to the number of elements transmitted.

To illustrate the use of DynamicArrays, we modify the physics_server **accept** statement given above.

```
/* In the physics interface definition */

DynamicArray  pressure_value  pressure_array;
DynamicArray  temperature_value  temperature_array;
DynamicArray  density_value  density_array;
                                        .
                                        .
                                        .
/* Physics_server declarations */

pressure_array  pressure;
temperature_array  pressure;
density_array density;
                                        .
                                        .
                                        .
/* The accept statement */

while(bound)
   {
   accept physics_client
       {
       on compute_density(gas, pressure, temperature)
           {
           if( pressure.len != temperature.len)
               {
```

```
                    give physics_client
                          {
                          density.len = 0;
                          compute_density(density);
                          }
                    }
              else
                    {

                    /* Note: the number of values is contained in both
                           pressure.len and temperature.len */

                    density_calculation(gas, pressure, temperature,
                           density);

                    give physics_client
                           {
                           compute_density(density);
                           }
                    } /* end of if-else */
              } /* end of on compute_density */
        } /* end of accept */
  } /* end of while */
```

When the ELROS generated code or the programmer allocates space from the heap for DynamicArrays, the programmer must free it before losing track of the values in “len” and “val”. To make this more convenient, the ELROS preprocessor generates free routines for DynamicArray types and for all types that contain one or more DynamicArray components (i.e., a Record or Union containing a DynamicArray component or a component that has a DynamicArray nested somewhere within it). Access to these free routines is possible by using the “rofreei()” and “rofree()” routines.

Rofreei() is called with the following parameters :

```
rofreei("type", &variable, interface)
```

The first argument is the ELROS type of the variable. Note that the type is placed within double quotes. The second argument is the address of the variable containing heap space to be freed. The third argument is the interface from which the type is drawn. Note that the interface name is not placed within double quotes. A call to rofreei() frees the space allocated either by the programmer or by the runtime system associated with "variable".

The ELROS runtime system must know the interface from which a type is drawn in order to call the correct routine to free its space (e.g., two interfaces imported into the embedded language code might define types of the same name). Rofreei() specifies the interface name through its third argument. The ELROS runtime system remembers this interface name in an internal variable, allowing a subsequent call of rofree() :

```
rofree("type", &variable)
```

The first argument is the ELROS type of the variable. Again, notice that the type is placed within double quotes. The second argument is the address of the variable containing heap space to be freed.

To illustrate how rofreei() and rofree() might be used, consider the variable "pressure" declared above. When the user sets the "len" and "val" fields of pressure, he normally would allocate space from the heap to hold the DynamicArray element values. After sending the values, he could free the space he allocated by calling :

```
rofreei(“pressure_array”, &pressure, physics);
```

If he subsequently received a value into "pressure", after using the values it references, he could free the space allocated from the heap by the ELROS runtime system by calling :

```
rofree(“pressure_array”, &pressure);
```

Eliminating the interface name is possible because he previously called rofreei().

On the server side, rofreei() and rofree() would be used after the **accept** statement is executed, e.g.,

```
while(bound)
    {
    accept physics_client
          {
          on compute_density(gas, pressure, temperature)
                                .
                                .
                                .
          } /* end of accept */

    rofreei("pressure_array", &pressure, physics);
    rofree("temperature_array", &temperature);
    rofree("density_array", &density);

    } /* end of while */
```

4.2 Records

ELROS also provides a way to define Record types. The ELROS Record type constructor syntax is very much like the struct syntax in C :

Record { *record-list* } *record-type-name*;

All Record type definitions begin with the keyword **Record**, followed by a semicolon-separated list of Record elements (record-list) contained within braces, and then by the name of the Record type, terminated by a semicolon. The simplest element definition is of the following form :

type-name   *element-name* ;

For example, suppose we wish to put the pressure and temperature Arrays of the compute_density operation in a structure, rather than making them independent parameters. The following interface specification accomplishes this :

```
interface physics;

   Integer(1)  gas_identifier;
   Real(12,9)  pressure_value;
   Real(14,7)  temperature_value;
   Real(48,9)  density_value;

   Array       pressure_value          pressure_array[20];
   Array       temperature_value       temperature_array[20];
   Array       density_value           density_array[20];

   Record
       {
       pressure_array          pressure_arg;
       temperature_array       temperature_arg;
       }  PT_record;

   call      compute_density      10
                  in  gas_identifier,  PT_record,
                  out density_array ;

end;
```

The client would then define a variable of type PT_record, initialize it and call the compute_density operation with the following **call** statement :

```
  PT_record  pressure_temperature;
                              .
                              .
                              .
```

```
compute_density@physics_server(gas,  pressure_temperature,
                                                density{n});
```

Notice that there is no length specified on the second parameter, since it is a Record and not an Array. Consequently, we have reintroduced the problem that we had before, namely, using the Record structure as we have defined it requires us to send all the elements in both the pressure and temperature Array to the server. One way to solve this problem is to define pressure_array and temperature_array as DynamicArrays.

However, there is another way to attack this problem – define the second parameter to the compute_density operation as an Array of Records, rather than a Record of Arrays[19]. The interface specification to do this is :

```
interface physics;

    Integer(1) gas_identifier;
    Real(12,9) pressure_value;
    Real(14,7) temperature_value;
    Real(48,9) density_value;

    Record
        {
        pressure_value          pressure;
        temperature_value       temperature;
        } PT_record;

    Array   PT_record           PT_array[20];
    Array   density_value       density_array[20];

    call    compute_density     10
                in gas_identifier, PT_array,
                out density_array ;
```

[19] Of course, we may not always be able to reorganize the parameter data structures in a manner similar to that shown here. However, in practice such restructuring is many times possible.

end;

This organization of the parameters to compute_density allows the number of elements in the second parameter, PT_array, to be given at runtime. The compute_density call would then look like :

```
PT_array        PT_args;
density_array   density;
int             n = 20;

                .
                .
                .
compute_density@physics_server(gas, PT_args{10}, density{n});
```

On the server side, Record parameters are handled in the same way as parameters of other types. For example, the **accept** statement of the compute_physics server that imports the above interface would be :

```
PT_array        PT_args;
density_array   density;
int             arg_count;
                .
                .
                .
while(bound)
  {
  arg_count= 20;

  accept physics_client
     {
     on compute_density(gas, PT_args{arg_count})
        {
        density_calculation(gas, arg_count, PT_args, density_array);
```

```
                give  physics_client
                      {
                      compute_density(density_array{arg_count});
                      }
                }
        }
}
```

Note there is no reason to test whether the number of pressure values equals the number of temperature values, since they are guaranteed the same by being in the same Record Array.

Records may contain Record, Array and Union definitions within their own definitions in addition to simple type declarations. We illustrate this feature in examples that appear below.

4.3 Unions

ELROS allows the definition of data types that are Unions of other types. A Union type is defined with the following Union type constructor :

Union {*union-list*} *union-type-name*;

The *union-list* consists of a set of Union element definitions with the following format :

type-name *element-name*;

We illustrate the use of Union types with the following modification to the physics interface. Sometimes it is desirable to compute the density of a gas composed of several elements. For example, we may be interested in the density of a gas mixture composed of hydrogen, helium and nitrogen at a particular pressure and temperature. This requires specifying the percentage of each gas in the mixture as well as the pressure and temperature.

We modify the compute_density operation so its first parameter may be either the atomic mass of a gas or an Array of Records, the first element of each Record specifying the gas and the second element specifying the percentage of that gas in the mixture. The physics interface specification becomes :

```
interface physics;

    Integer(1) gas_identifier;
    Real(12,9) pressure_value;
    Real(14,7) temperature_value;
    Real(48,9) density_value;

    Record
        {
        gas_identifier element;
        Real(10,8) percentage;
        } gas_percentage;

    Array gas_percentage GP_array[20];

    Union
        {
        gas_identifier simple_gas;
        Record
            {
            gas_identifier number; /* number of components */
            GP_array component;
            } mixture;
        } Gas;

    Record
        {
        pressure_value          pressure;
        temperature_value       temperature;
        } PT_record;

    Array    PT_record          PT_array[20];
```

```
    Array    density_value          density_array[20];

    call     compute_density      10
                     in Gas, PT_array,
                     out density_array ;

end;
```

Union types in C are non-discriminated. That is, there is no information available at runtime that specifies which member is held within the union. ELROS Union types, on the other hand, must be discriminated, since there must be a way to convey the identity of the Union member to a program that executes a **give** or an **accept** statement. Consequently, ELROS Unions map to a C struct with two members – one the union specified in the ELROS type and the other a tag that specifies which member is present in the union. The ELROS Union type Gas results in the following C struct type :

```
typedef
    struct
            {
            int tag;
            Union
                {
                gas_identifier  simple_gas;
                struct
                        {
                        gas_identifier  number;
                        GP_Array  component;
                        } mixture;
                } un;
            } Gas;
```

When a variable of type Gas is an **in** parameter appearing in a **call** statement or a parameter appearing in a **give** statement, the discriminator "tag" must be set by the programmer to the identifier of the Union member to send. When a variable of type Gas is a parameter appearing in an **accept** statement, the discriminator "tag" is set by the ELROS runtime system to the value corresponding to the Union member received.

Elements are numbered consecutively beginning at 0. For example, "tag" is set to 0 to select the Union member "single_gas" and to 1 to select the Union member "mixture". Referencing a Union member requires naming the "un" identifier in the dotted construction corresponding to the member name. For example, to reference the "number" member of the Record "mixture" in a variable Gas_arg defined as type Gas requires the dotted construction :

```
Gas_arg.un.mixture.number
```

Given the physics interface defined above, the following code requests the physics_server to calculate the density of a mixture of gases consisting of 50% hydrogen, 40% helium and 10% nitrogen :

```
PT_array       PT_args;
Gas            Gas_arg;
density_array  density;
int            n = 20;

Gas_arg.tag = 1;
Gas_arg.un.mixture.number = 3;
Gas_arg.un.mixture.component[0].element = 1;
Gas_arg.un.mixture.component[0].percentage = .5;
Gas_arg.un.mixture.component[1].element = 2;
Gas_arg.un.mixture.component[1].percentage = .4;
Gas_arg.un.mixture.component[2].element = 7;
Gas_arg.un.mixture.component[2].percentage = .1;
                    .
                    .
                    .
compute_density@physics_server(Gas_arg, PT_args{10}, density{n});
```

The **accept** statement in the physics server would look like :

```
PT_array          PT_args;
density_array     density;
Gas               Gas_arg;
int               arg_count ;
                              .
                              .
                              .
while(bound)
    {
    arg_count= 20;

    accept physics_client
          {
          on compute_density(Gas_arg, PT_args{arg_count})
                {

                /* test tag to see what type of gas (simple or mixture) */

                if(Gas_arg.tag == 0)
                      {
                      density_calculation_simple(Gas_arg.un.single_gas,
                                                 arg_count, PT_args, density);
                      }
                else /* Gas_arg.tag == 1 */
                      /* note : Gas_arg.tag MUST equal either 0 or 1. If not
                            an ELROS runtime system internal error would
                            occur */

                      {
                      density_calculation_mixture(Gas_arg.un.mixture,
                                                 arg_count, PT_args, density);
                      }

                /* return the density Array */

                give physics_client
```

```
                {
                compute_density(density{arg_count});
                }
            }
        }
    }
```

4.4 Exception Handling

So far, we have made little mention of errors and the facilities in ELROS to handle them. The experienced distributed application programmer knows that error handling is one of the most troubling aspects of his craft. The error modes that arise in distributed systems include those that occur in stand-alone systems as well as others that are significantly harder to correct.

We have already seen the basic mechanism provided by ELROS for error handling in another context, i.e., the **exceptions** clause. The **exceptions** clause can be attached to the **bind_addr** statement, the **call** statement, the **accept** statement, the **give** statement and other statements not covered in this primer. We have used the **exceptions** clause to detect the unbinding of an instance using the on-done clause. However, there are other on-clauses that may appear within the **exceptions** clause, specifically (in the following definitions square brackets are used to show optional fields; they are not literal text) :

on com_error(*status* [, *string-variable*]) *statement*

When a communications error occurs during the execution of the statement to which the **exceptions** clause is attached, control passes to the **com_error** on-clause. An example of a **com_error** is a timeout on a **bind_addr** or **call** statement. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the communications error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *statement* given after the *status* or *string-variable* is executed when the **com_error** on-clause gains control.

on internal_error(*status* [, *string-variable*]) *statement*

If an ELROS runtime system error occurs, control passes to the **internal_error** on-clause. This error indicates that the runtime system is faulty. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the internal error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *statement* given after the *status* or *string-variable* is executed when the **internal_error** on-clause gains control.

on protocol_error(*status* [, *string-variable*]) *statement*

When an error in the underlying Remote Operations protocol occurs, control passes to the **protocol_error** on-clause. This error indicates that the remote operations service element of the correspondent is faulty. The first parameter in parentheses, *status*, is an integer variable which receives the numerical code of the protocol error. The second optional parameter, *string-variable*, is a string variable capable of storing at least 512 characters. This variable will receive a textual description of the error. The *statement* given after the *status* or *string-variable* is executed when the **protocol_error** on-clause gains control.

on missing_op(*string-variable*) *statement*

When an **accept** statement receives an operation for which no on-clause is specified, control passes to the **missing_op** on-clause. The first parameter, *string-variable*, is a string variable capable of storing at least 32 characters. This variable will receive the operation name, if the operation is defined for the interface. Otherwise, it indicates that the operation is unknown in the interface. The *statement* given after the *string-variable* is executed when the **missing_op** on-clause gains control.

on missing_arg(*op-string-variable*, *arg-string-variable*) *statement*

When an **accept** statement receives an argument within an operation for which no parameter is specified, control passes to the **missing_arg** on-clause. The first parameter, *op-string-variable*, is a string variable capable of storing at least 32 characters. This variable will receive the operation name. The second parameter, *arg-string-variable*, receives the identifier of the argument (its position in the case of operations with positional parameters, or its name in the case of operations with named parameters) if it is defined for the operation. Otherwise, an indication that the argument is not valid for the operation is given. The *statement* given after the *op-string-variable* and *arg-string-variable* is executed when the **missing_arg** on-clause gains control.

on bad_op_id(*op_id*) *statement*

When a **call** statement executes, an operation request is sent to the server. This request is tagged by the **call** statement with an operation identifier. When the results of the **call** are returned to the server, the code generated by the ELROS preprocessor examines the operation identifier of the returned results, ensuring that the returned operation identifier equals the operation identifier sent. If the two operation identifiers are unequal, a **bad_op_id** exception is generated and the **bad_op_id** on-clause receives control. The *op_id* argument (a long integer) receives the value of the op_id returned by the server and the *statement* following it is executed.

We illustrate the use of **exceptions** on-clauses with the following example. We normally would be interested in handling communication errors that occur during the execution of a **call** statement. Attaching an **exceptions** clause with an explicit **com_error** on-clause to the **call** statement in our example would result in the following code :

```
int error = 0;
char text[512];
                    .
                    .
                    .
compute_density@physics_server(Gas_arg, PT_args{10}, density{n});

exceptions
    {
    on com_error(&error, text)
        {
        unbind(physics_server);
        printf(" communications error - %s", text);
        return(error);
        }
    }
```

Attaching an **exceptions** clause to the **accept** statement in the physics server would result in the following code :

```
int error = 0;
char text[512];
                    .
                    .
                    .
  accept physics_client
      {
      on compute_density(Gas_arg, PT_args{arg_count})
          {
                    .
                    .
                    .
          }
      }
  exceptions
```

```
{
on com_error(&error, text)
      {
      unbind(physics_client);
      printf(" communications error - %s", text);
      return(error);
      }
}
```

The ELROS preprocessor supplies default **exception** on-clauses for all those not specified by the programmer. So, for example, in a program that contains the above code, a **missing_op** or **missing_arg** error would cause an exception and control would transfer to the default exception code for those errors. The default handlers provided by ELROS all cause the program to exit.

4.5 Defining and using named parameters

Operations defined using named parameters are useful when the application programmer wishes to elide (i.e., leave out) some parameters on a particular operation call[20]. This might be desirable when some of the parameters contain a large amount of data and that data was already transferred to the server in a previous operation.

In order to discuss this possibility, we need to explain the difference between a stateless and a state-retaining server. A stateless server accepts operations, acts on them and then discards the values associated with the operation before waiting for a new operation. A state-retaining server accepts operations, acts on them, but does not discard the values associated with the operation. Instead, it waits for a new operation and substitutes the values given in a previous operation for any parameters that are unspecified.

---

[20] It is possible to elide parameters in operations using positional parameters as well. However, the syntax, which simply leaves out the parameter in the comma separated parameter list, is hard to read and can easily lead to errors, such as leaving out the wrong parameter. Named parameter lists provide a textually obvious way of eliding parameters.

For example, let's add a second operation to the physics interface. Instead of computing a density from a pressure and temperature, the new operation computes a pressure from a temperature and a density.

Now suppose a client first calls compute_density, specifying a set of pressures and temperatures, and then wants to specify a new set of temperatures for the given densities (e.g., the star warms up for some reason) in order to compute the corresponding new set of pressures.

If the server is stateless, the client program would call compute_density, receiving back a set of densities, and then call compute_pressure, providing the densities it just received from the server as a parameter, along with the new temperatures parameter. However, if the server is state-retaining, the client could call compute_density and then compute_pressure, specifying in the latter call only the new set of temperatures. On the call to compute_pressure, the client need specify neither the composition of the gas nor the set of densities, since the server would retain these from the previous compute_density call. We say the gas and density parameters to compute_pressure are elided when it is called.

The novice distributed application programmer might make the following observation. If we define compute_pressure to take only a set of temperatures, then eliding the gas and density parameters would be unnecessary. The problem with this approach is that you now **must** call compute_density before calling compute_pressure. In some other application that uses the physics server, you could not call compute_pressure independently.

Operations with named parameter lists are defined exactly like those with positional parameter lists, except the operation parameters are first defined in a “names list”. These names are then used to specify parameters. Each name may be used only once in a given operation parameter list.

We will define the two physics server operations using named parameter lists. To do this, we first must define the names. This is done with a **names** statement that begins with the reserved word **names** followed by a list of comma separated triples and terminates with a semicolon. The triples have the format :

datatype *identifier* *integer*,

The *identifier* is the name used subsequently within the interface specification, the *datatype* is its type and the *integer* is a positive value associated with the name that is unique for names within the interface. For example, the parameter list for "compute_density" could be defined as a named parameter list containing four names : 1) GAS, 2) PRESSURE, 3) TEMPERATURE and 4) DENSITY[21] . The "compute_density" operation is then defined by replacing the parameters specified by their types in the positional parameter list with the corresponding names defined in the names statement. To keep compute_density and compute_pressure compatible, we use a separate temperature and pressure Array parameter in both operations. We define "compute_density" as a named operation with the following interface definition:

```
interface physics;

    Integer(1)  gas_identifier;
    Real(12,9)  pressure_value;
    Real(14,7)  temperature_value;
    Real(48,9)  density_value;

    pressure_value          pressure;
    temperature_value       temperature;

    Record
        {
        gas_identifier  element;
        Real(10,8)  percentage;
        }  gas_percentage;

    Array  gas_percentage  GP_array[20];
```

[21] Although we show the names in the namelist in all capitals, this is done merely to make them stand out. There is no constraint on the composition of names other than they must be valid identifiers in C.

```
    Union
          {
          gas_identifier  simple_gas;
          Record
                {
                gas_identifier  number;
                GP_array  component;
                } mixture;
          } Gas;


    Array    pressure          pressure_array[20];
    Array    temperature       temperature_array[20];
    Array    density_value     density_array[20];

  names
      DENSITY             density_array           1,
      GAS                 Gas                     2,
      TEMPERATURE         temperature_array       3,
      PRESSURE            pressure_array          4;

    call     compute_density        10
             in GAS, TEMPERATURE, PRESSURE,
             out DENSITY;

    call     compute_pressure       11
             in GAS, TEMPERATURE,  DENSITY,
             out PRESSURE;

end;
```

Note that the order in which the names appear in the **names** statement bears no relation to the order in which the names may appear in the definition of an operation.

Using operations with named parameters in **call** statements requires specifying both the name of a parameter and its value, separated by a colon.

To elide a parameter, simply omit the name-value pair. For example, the following code calls the compute_density operation defined with named parameters :

```
temperature_array        temperature;
pressure_array           pressure;
int density_count = 20;
int pressure_count = 20;
                    .
                    .
                    .
compute_density@physics_server(GAS:Gas_arg,
                               TEMPERATURE: temperature{10},
                               PRESSURE:pressure{10},
                               DENSITY:density{density_count});
```

Calling the compute_pressure operation and eliding both the GAS and DENSITY parameters is achieved by :

```
/* initialize temperature */
                    .
                    .
                    .
compute_pressure@physics_server(TEMPERATURE:temperature{10},
                                PRESSURE:pressure{pressure_count});
```

Calling the compute_pressure operation without eliding any of its parameters would look like :

```
compute_pressure@physics_server(TEMPERATURE:temperature{10},
                                GAS:Gas_arg,
                                DENSITY:density{10},
                                PRESSURE:pressure{pressure_count});
```

Notice that the parameters to an operation with a named parameter list may appear in any order since the parameter name discriminates between them.

When operations with named parameter lists appear in **accept** statements, the name of the parameter precedes a variable of the appropriate type, again separating the name from the variable by a colon. For example the physics server **accept** statement for the two physics interface operations might look like the following :

```
int pressure_count = 20;
int temperature_count = 20;
int density_count = 20;

/* initialize the size variables, Gas_arg, PT_args and temperature to
     default values in case they are elided by the client */
                              .
                              .
                              .
     accept physics_client
          {
          on compute_density(GAS:Gas_arg,
                         TEMPERATURE:temperature{temperature_count},
                         PRESSURE:pressure{pressure_count});
               {
                              .
                              .
                              .
               give physics_client
                    {
                    compute_density(DENSITY:density{pressure_count});
                    }
               }
          on compute_pressure(GAS:Gas_arg,
                         TEMPERATURE:temperature{temperature_count},
                         DENSITY:density{density_count})
               {
```

```
                    .
                    .
                    .
          give physics_client
               {
               compute_pressure(PRESSURE:pressure{density_count});
               }
          }
     }
exceptions
     {
     on com_error(error, text)
          {
          unbind(physics_client);
          printf(" communications error - %s", text);
          return(error);
          }
     on done
          {
          unbind(physics_client);
          bound = FALSE;
          }
      }
```

This **accept** statement could be used both in a stateless server as well as a state-retaining server. The difference between these two types of server is how they initialize the variables that are the parameters to the operations. For example, if we have written four routines (init_Gas_arg, init_temperature, init_pressure, and init_density) that initialize Gas_arg, temperature, pressure and density to default values, the following code implements a stateless server.

```
short bound = TRUE;
          .
          .
          .
```

```
while(bound)
    {

    init_Gas_arg(&Gas_arg);
    init_temperature(temperature);
    init_pressure(pressure);
    init_density(density);

    int pressure_count = 20;
    int temperature_count = 20;
    int density_count = 20;

    accept  physics_client
          {
          on  compute_density(GAS:Gas_arg,
                          TEMPERATURE:temperature{temperature_count},
                          PRESSURE:pressure{pressure_count})
                          {
                                  .
                                  .
                                  .
                           }
          on  compute_pressure(GAS:Gas_arg,
                          TEMPERATURE:temperature{temperature_count},
                          DENSITY:density{density_count})
                          {
                                  .
                                  .
                                  .
                           }
            }
    exceptions
            {
            on  com_error(error,  text)
                           {
                                  .
```

```
                    .
                    .
                }
        on done
                {
                    .
                    .
                    .
                }
        }

    }
```

Each time through the server dispatch loop, the parameters are initialized to their default value. Consequently, the server does not save the parameter values between operation requests. On the other hand, the following code implements a state-retaining server :

```
init_Gas_arg(&Gas_arg);
init_temperature(temperature);
init_pressure(pressure);
init_density(density);
short bound = TRUE;

bind_addr(physics_client,  ...);
while(bound)
    {

    pressure_count = 20;
    temperature_count = 20;
    density_count = 20;

    accept  physics_client
          {
          on  compute_density(GAS:Gas_arg,
```

```
                        TEMPERATURE:temperature{temperature_count},
                        PRESSURE:pressure{pressure_count})
                        {
                                .
                                .
                                .
                         }
        on  compute_pressure(GAS:Gas_arg,
                        TEMPERATURE:temperature{temperature_count},
                        DENSITY:density{density_count})
                        {
                                .
                                .
                                .
                         }
         }
    exceptions
         {
         on com_error(error, text)
              {
                                .
                                .
                                .
              }
         on done
              {
                                .
                                .
                                .
              }
         }

    }
```

This code initializes the parameter variables before the server dispatch loop is entered and retains their values across operation calls. The server loop exits when the client unbinds the instance (i.e., when the **done** on-clause of the attached **exceptions** clause is given control) at which point the server would iterate through an outer loop (not shown), reinitialize the operation parameters and enter the dispatch loop when the **bind_addr** unblocks.

If a client requests a compute_density operation, which fills in the Gas_arg parameter and changes the server's density Array, and then requests a compute_pressure operation without specifying either the Gas_arg or the density Array, the compute_pressure on-clause in the **give** statement completes, but does not disturb the values in either Gas_arg or density. Thus, when the semantics of the compute_pressure operation execute (i.e., the code not shown, but symbolized by three vertical dots), the values of these parameters are those when the compute_density operation completed. In this way the client can economize the amount of data it transmits to the server on each operation call. For our simple example, this does not save much. But for a server with operations that potentially take very large Arrays of data (say, 30,000 Reals), the use of a state-retaining server can greatly increase the efficiency of calculations over that obtainable with a stateless server.

4.6 Asynchronous Operations

In the examples we have covered so far, it has been acceptable for clients to execute a remote operation and then block waiting for a result. ELROS provides the **call** statement for this style of programming. There are certain situations, however, for which the synchronous nature of the **call** statement is inappropriate.

For example, suppose the client has a requirement to perform several operations in a row, say a compute_density operation followed immediately by a compute_pressure operation. One way of accomplishing this is to use two **call** statements.

However, this causes the client to block waiting for the compute_density operation to complete before sending the compute_pressure operation. Execution of the compute_pressure operation is delayed while the result of the compute_density operation is communicated back to the client and the compute_pressure operation is communicated to the server.

To eliminate this inefficiency, ELROS provides a way to communicate a set of operations at the same time so the server can begin execution of an operation as soon as it completes execution of the previous operation. This is known as **batching** (which is a special case of asynchronous operations).

Another situation in which the **call** statement is unsuitable occurs when the client has a requirement to compute in parallel with the server and yet maintain the sequentiality of its remote service requests[22]. For example, suppose a relaxation method is used to calculate the stable values of temperature, pressure and density for a volume of gas under given boundary conditions (fig. 5). It requests the server to compute pressures given the corresponding temperatures and densities (**1**). Concurrently, it computes new estimates of the temperatures based on previous temperatures and densities (**2**). After compute_pressure completes, the client calls compute_density with the new temperatures, utilizing the pressures left at the server by the compute_pressure operation (**3**). Then it computes new temperatures (**4**) for the next iteration (**5** & **6**).

Without the possibility of asynchronously requesting compute_density and compute_pressure operations, the compute_pressure operation at **1** and the compute_density operation at **3** would block the client, preventing it from computing the new temperatures at **2** and **4** in parallel with the server. One way to solve this blocking problem without asynchronous operations is to use a thread to perform the computations at **2** and **4**, allowing a second thread that calls compute_pressure and compute_density to block. However, this requires communications and synchronization between the two threads, greatly increasing the complexity of the client. That is, the first thread must communicate the old temperatures and pressures, or old temperatures and densities to the second thread, and the second thread must communicate the new temperatures to the first thread. Since the first thread will block waiting for the compute_pressure and compute_density operation to complete, the first thread must synchronize with the second thread in order to determine when the new temperatures are available. Consequently, providing a way to

[22] If there is no requirement to ensure operations made by the client to the server are in sequence, clients can use multi-threading to compute in parallel with the server. This does not require the use of asynchronous operations.

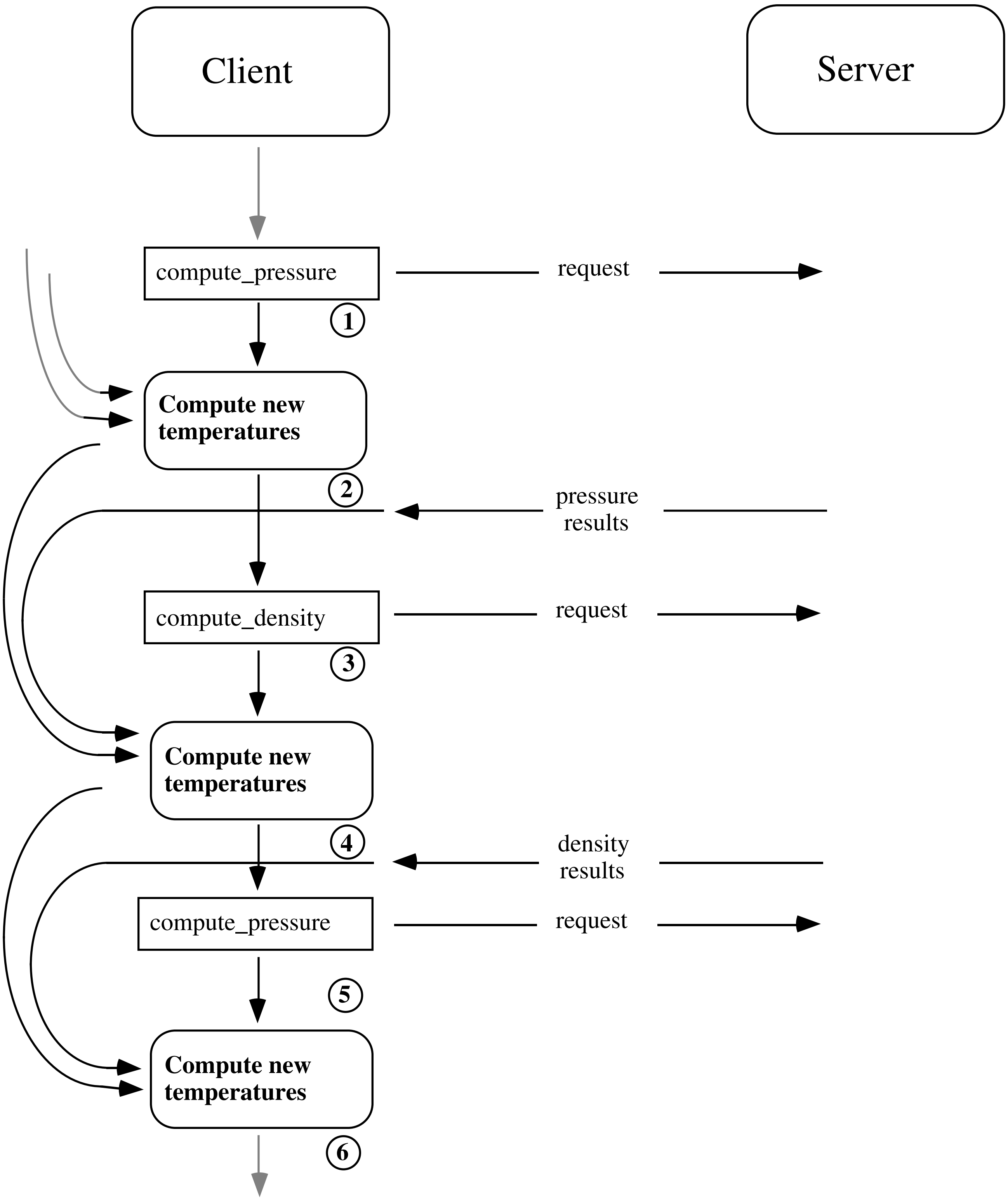

Figure 5 - Overlapping asynchronous operations

send an operation to a server without blocking is a useful feature of languages supporting distributed application programming. ELROS provides a way to send an operation to a server without blocking and then allows the client to wait for the result of that operation.

Asynchronous operations are defined much like synchronous operations, except they are prefixed by different keywords in their definition. An operation representing a service request from a client to a server is defined using the reserved word **invoke**. For example, the definition of asynchronous versions of the compute_density and compute_pressure operations is:

```
interface physics;

    pressure_value          pressure;
    temperature_value       temperature;

    Record
        {
        gas_identifier element;
        Real(10,8) percentage;
        }  gas_percentage;

    Array  gas_percentage  GP_array[20];

    Union
        {
        gas_identifier  simple_gas;
        Record
            {
            gas_identifier  number;
            GP_array  component;
            }  mixture;
        } Gas;

    Array    pressure          pressure_array[20];
    Array    temperature       temperature_array[20];
```

```
    Array    density_value         density_array[20];

  names
    density_array          DENSITY          1,
    Gas                    GAS              2,
    temperature_array      TEMPERATURE      3,
    pressure_array         PRESSURE         4;

    invoke   compute_density        10
             GAS, TEMPERATURE, PRESSURE;

    invoke   compute_pressure       11
             GAS, TEMPERATURE, DENSITY;

end;
```

Notice that the only difference between this interface specification and the one given in section 4.5 is the use of the reserved word **invoke** instead of the reserved word **call** and that the parameters to the asynchronous versions of these operations are the **in** parameters of the corresponding **call** versions.

Defined in this manner, the compute_density and compute_pressure operations represent one-way operations from the client to the server. To receive the results of these operations requires defining new operations representing one-way operations from the server to the client. This is achieved by defining these operations with the **result** reserved word, specifying the **out** parameters of the corresponding **call** operation.

For example, let's represent the return of results from the compute_density operation by the density_result operation and the return of results from the compute_pressure operation by the pressure_result operation. The definition of these operations would appear as :

```
interface physics;
                              .
                              .
                              .
```

```
    result   density_result          10
             DENSITY;

    result   pressure_result         11
             PRESSURE;
                            .
                            .
                            .
end;
```

Note that these new **result** operations have names different from their **invoke** operation cognates and that their parameters are the **out** parameters of the corresponding **call** operations. Also note that their operation numbers are the same as compute_density and compute_pressure. Operation numbers must be unique only within their class of operations, i.e., **invoke**, **result** and **error** (see below for a description of **error**). When determining the uniqueness of operation numbers, **call** operations are considered to belong to both the **invoke** and **result** classes.

We have already seen the ELROS statements that allow us to use these new asynchronous operations. To communicate an **invoke** operation from a client to a server or a **result** operation from a server to a client requires placing the operation in a **give** statement. A server waits for an **invoke** operation and a client waits for a **result** operation by using an on-clause in an **accept** statement.

However, there is a new wrinkle introduced by the asynchronicity of these operations. Since several operations may be invoked before a result is accepted, there must be some way for the client to determine that a received **result** corresponds to a given **invoke** operation.

To support this determination, ELROS tags an operation with an operation identifier[23] when it is sent in a **give** statement. An operation

---

[23] The Remote Operations Service standard on which ELROS is based calls these invoke identifiers. However, we find this confusing, since invoke identifiers appear on result and error operations as well. Consequently, we have chosen to call them operation identifiers (or operation ids).

identifier is an integer value[24] drawn from an internal ELROS runtime system variable associated with the instance. The runtime system increments the variable each time an operation is sent. This ensures that the next operation is tagged with a new identifier. The internal variable is initialized to 1 when the instance is bound by **bind_addr** .

A server obtains the identifier associated with an operation by including a long integer pointer variable[25] in square brackets after the reserved word **accept**. When the **accept** statement executes as the result of receiving an operation, the pointer is set to the address of a runtime system variable containing the operation identifier.

Tagging a **result** operation with the operation identifier of the corresponding **invoke** operation is accomplished by including a long integer in square brackets after the operation name within a **give** statement. This operation identifier is carried by the **result** and made available to the client through the **accept** statement mechanism described above (i.e., a long integer pointer appearing in square brackets after the reserved word **accept** receives the address of a runtime variable containing the operation id).

We now illustrate the use of asynchronous operations. To make an asynchronous compute_density request, given the above physics interface definition, the client would use the following code :

```
give  physics_server
    {
    compute_density(GAS:Gas_arg,
                      TEMPERATURE: temperature,
                      PRESSURE:pressure);
    }
```

To batch a compute_density and compute_pressure operation into one request requires the following code :

```
give  physics_server
```

---

[24] An advanced feature of ELROS allows these tags to be ASN.1 object identifiers. This feature is beyond the scope of the primer.

[25] Actually, any expression that evaluates to a long pointer lvalue may be specified.

```
{
compute_density(GAS:Gas_arg,
                TEMPERATURE: temperature,
                PRESSURE:pressure);
compute_pressure(TEMPERATURE:temperature);

}
```

It would also be possible to execute two **give** statements, one containing a compute_density operation and the other a compute_pressure operation. This would have the same effect, except that ELROS attempts to pack both operations in the same communications packet when operations are contained in the same **give** statement. If they are in separate **give** statements, the operations will be sent in different communications packets.

To receive the results of these asynchronous operations, the client executes an **accept** statement with on-clauses for the **result** operations. For example, the following code would accept results from either the compute_density or compute_pressure asynchronous operations :

```
int  density_count;
int  pressure_count;
int i;
long  *op_id;

short error = FALSE;
                        .
                        .
                        .
init_Gas_arg(&Gas_arg);
init_temperature(temperature);
init_pressure(pressure);
init_density(density);

bind_addr(physics_server,  ...)
```

```
for(i=1; i<3 && error == FALSE; i++)
    {
      density_count = 20;
      pressure_count = 20;

      accept [op_id] physics_server
            {
            on density_result(DENSITY:density{density_count});

                /* check op_id to ensure this result corresponds to the
                    appropriate operation that was invoked. Operations are
                    tagged starting at 1. The programmer can check the
                    op_id by keeping track of the operations that have been
                    sent to the server, using an integer variable incremented
                    each time an operation is sent or by some other means.*/

                if(*op_id != i)
                    {
                    unbind(physics_server);
                    printf("mismatch between invoke and result op id");
                    error = TRUE;

            on pressure_result(PRESSURE:pressure{pressure_count})

                /* check op_id to ensure this result corresponds to the
                    appropriate operation that was invoked. Operations are
                    tagged starting at 1.  The programmer can check the
                    op_id by keeping track of the operations that have been
                    sent to the server, using an integer variable incremented
                    each time an operation is sent or by some other means.*/

                if(*op_id != i)
                    {
                    unbind(physics_server);
                    printf("mismatch between invoke and result op id");
                    error = TRUE;
```

```
                }

            }
        exceptions
            {
                                    .
                                    .
                                    .
            on done
                {
                /* this is an error. Unbind and report it */

                unbind(physics_server);
                printf("unexpected disconnect by the physics server");
                error = TRUE;
                }
            } /* end of exception */
    } /* end of for loop */
```

Note the similarity between this **accept** statement code fragment and the one shown previously for the physics server (with the exception that op_id is used to check **invoke/result** operation synchronization). Although the **for** statement is not actually necessary (since the physics server will finish the compute_density operation, send its result to the client and then execute the compute_pressure operation), it simplifies the code. Without using the **for** loop, we would need to include two **accept** statements, one containing an on-clause for the density_result operation, the other containing an on-clause for the pressure_result operation, each attaching its own **exceptions** clause. The for loop allows us to compact these two **accept** statements into one.

The code in the physics server to accept and process asynchronous operation requests is almost identical to that used for **call** operations :

```
int  pressure_count;
int  temperature_count;
int  density_count;
```

```
long  *op_id;

/* initialize Gas_arg, PT_args and temperature to default values in case
     they are elided by the client and bind "physics_client" */
                                        .
                                        .
                                        .
short bound = TRUE;

while(bound)
     {
     pressure_count = 20;
     temperature_count = 20;
     density_count = 20;

     accept  [op_id]  physics_client
            {
            on  compute_density(GAS:Gas_arg,
                                   TEMPERATURE:
                                          temperature{temperature_count},
                                   PRESSURE:pressure{pressure_count});
                 {
                                        .
                                        .
                                        .
                 give  physics_client
                        {
                        density_result  [*op_id]  (DENSITY:density{pressure_count});
                        }
                 }
            on  compute_pressure(GAS:Gas_arg,
                                     TEMPERATURE:
                                          temperature{temperature_count},
                                     DENSITY:density{density_count})
                 {
                                        .
```

```
                    .
                    .
        give  physics_client
              {
              pressure_result  [*op_id]  (PRESSURE:pressure{density_count});
              }
        }
    }
exceptions
    {
                    .
                    .
                    .
    }
}
```

The main difference between this physics server dispatch loop and the one given in section 4.5 is that the **give** statement contained in each operation on-clause within the **accept** statement references the asynchronous **result** operation (i.e., density_result and pressure_result) instead of a synchronous **call** operation. It also provides the op_id received in the **accept** statement as a tag for the **result** operation.

The preceding example illustrates how a client can batch asynchronous operations. We now illustrate how to implement the code schema shown in figure 5. To accomplish this, no change to the server dispatch loop is necessary. This is desirable because the server must execute remote operations for a variety of clients, some of which may be batching operations, others making single operation requests, and others carrying out the interlaced execution of remote operations we are now about to illustrate.

The interlaced client code corresponding to the strategy illustrated in figure 5 is as follows :

```
/* declaration of Gas_arg, temperature, pressure, density,
    density_count, pressure_count, and i */
                    .
```

```
                    .
                    .
init_Gas_arg(&Gas_arg);
init_temperature(temperature);
init_pressure(pressure);
init_density(density);

bind_addr(physics_server,  ...)
                    .
                    .
                    .
/* request initial compute_pressure and compute_density operations
      that transfer the GAS, PRESSURE and DENSITY arguments to the
      server, where they are retained. (Note that operations sent
      subsequent to these first two will begin with an operation identifier
      of 3.)  Then compute the new temperatures that start the iteration.
      */
                    .
                    .
                    .
long  sent_op_id,  *received_op_id;

sent_op_id = 2;

for(i=1; i<NUMBER_OF_ITERATIONS; i++)
      {
      density_count = 20;
      pressure_count = 20;

      give  physics_server
            {
            compute_pressure(TEMPERATURE:  temperature{10});
            sent_op_id++;
            }
      exceptions
            {
```

```
                    .
                    .
                    .
    }

/* compute new temperatures from the densities received in the
    previous iterations of the loop and from the existing
    temperatures */
                    .
                    .
                    .
/* now wait for the new pressures to arrive */

accept [received_op_id] physics_server
    {
    on pressure_result(PRESSURE:pressure{pressure_count});

    /* check to ensure the op_id on the result is correct */
         if(*received_op_id != sent_op_id)
           {
           unbind(physics_server);
           printf("mismatch between invoke and result op id");
           exit(0);
           }
                    .
                    .
                    .
    }
exceptions
    {
                    .
                    .
                    .
    }
```

```
/* use the newly calculated pressures (which are already at the
       server, so they need not be sent) and the new temperatures to
       calculate new densities */

give physics_server
       {
       compute_density(TEMPERATURE:temperature{10}});
       sent_op_id++;
       }

/* use pressure results and existing temperatures to compute new
       temperatures */
                              .
                              .
                              .
/* wait for the new densities to arrive, then iterate*/

accept [received_op_id] physics_server
       {

       on density_result(DENSITY:density{density_count});

       /* check to ensure the op_id on the result is correct */
               if(*received_op_id != sent_op_id)
                   {
                   unbind(physics_server);
                   printf("mismatch between invoke and result op id");
                   exit(0);
                   }
                              .
                              .
                              .
       }
exceptions
       {
                              .
```

.

.

}

}

4.7 Handling Errors Specific To An Application

In section 4.4 we discuss how an application handles errors generated by the ELROS runtime system. There is another class of errors with which an application must deal. These are the errors generated by the application itself. For example, when the physics server accepts either a compute_density or compute_pressure operation, it must check to ensure that the description of a mixture of gases specifies a set of percentages that add up to 100%. If this is not true, it should return an error indication to the client.

Up to this point, we have been using the convention of returning a result with no entries to signify an application error (e.g., the compute_density operation returns a density Array with zero elements when it detects an error). This sort of convention has the disadvantage that the specific nature of the error is not communicated. For example, the server should return an error indication when an invalid gas element identifier is specified and when the number of elements in a gas mixture equals zero as well as when the percentages of elements in a gas mixture do not add to 100%. With the convention we have so far adopted, it is not possible to discriminate between these classes of error.

ELROS provides a way to signal an error without using an application dependent convention. In the previous section we described how to define both **invoke** and **result** operations that can be used to implement asynchronous service requests. There is another type of operation used to communicate application errors - **error** operations[26].

Error operations are returned from a server to a client when an application-specific error occurs. They are transmitted using **give**

---

[26] There is no way to use an **error** operation to signal an application error occurring on a **call** operation. There must be a parameter of the **call** operation used for this purpose.

statements and received using **accept** statements just like **invoke** and **result** operations. **Error** operations take a set of parameters that allow a server to provide more detailed information about the error.

The definition of an **error** operation appears in an interface specification. An **error** operation is defined by writing the reserved word **error** followed by the name of the error, an integer identifier, and the parameters that will be transmitted by the server when the error occurs. For example, suppose we are defining the errors that may occur as the result of the compute_density operation. Each **error** operation indicates that there was something wrong with the parameters given by the client in the operation invocation. Three parametric problems are recognized : 1) an invalid gas element was specified (e.g., the element wasn't a gas), 2) the percentages of component gases in a mixture do not add up to 100%, and 3) the number of elements specified for a mixture of gases equals zero.

For each of these problems we define in the physics interface a corresponding **error** operation as follows :

```
interface physics;
                        .
                        .
                        .
    error    invalid_element_error     10
             gas_identifier;

    error    percentage_error          11
             Real(10,8);

    error    no_elements_error         12;
                        .
                        .
                        .
end;
```

Invalid_element_error takes one argument that specifies the atomic number of an element that is not a gas in a proposed mixture. The argument of percentage_error specifies the total percentage (not equal to

100%) given in the associated compute_density or compute_pressure call. The no_elements_error takes no argument. In order to use **error** operations defined in an interface, the application protocol designer must use the **give/accept** model of remote service request (i.e., there isn't a way for a client to obtain application specific **error** operations generated by the server when the client uses a **call** statement). The following example illustrates how an asynchronous compute_density operation might be called by the client and how the client would provide for the receipt of the three possible compute_density errors.

```
gas_identifier element;
Real(10,8) percentage;
                    .
                    .
                    .
give physics_server
      {
      compute_density(GAS:Gas_arg,
                      TEMPERATURE: temperature{10},
                      PRESSURE:pressure_new{10})
                      {
                          .
                          .
                          .
                      }
      }
                    .
                    .
                    .
accept physics_server
      {
      on density_result(DENSITY:density{density_count})
                      {
                          .
                          .
                          .
```

```
                    }
on  invalid_element_error(element)
                    {
                          .
                          .
                          .
                    }
on  percentage_error(percentage)
                    {
                          .
                          .
                          .
                    }
on  no_elements_error
                    {
                          .
                          .
                          .
                    }

}
                          .
                          .
                          .
```

4.8 Using Indefinite Length String Types

In most scientific work the use of definite length string types, i.e., those that specify an integer value in parentheses after the base type name, is sufficient when writing distributed applications. However, there are situations in which the programmer must use the indefinite length form. Indefinite length string variables, i.e., those whose type is one of the indefinite length string types, behave somewhat differently in **give** and **accept** statements than their definite length cognates.We concentrate on the difference between definite and indefinite length character strings, since the other string types (i.e., **BitString**, **OctetString** and **ObjectIdentifier**) are unlikely to be used in scientific applications.

When a definite length LenTerm character string variable is given as the parameter to an operation in the on-clause of a **give** statement, the ELROS generated code transmits as many characters as the “len” field indicates. If the variable is one of the 4 NULTerm character string types, the ELROS generated code transmits characters from the string until a NUL character is encountered. If a NUL character is not encountered in processing the n+1 characters of the array, an exception is generated.

When using an indefinite length variable as the parameter to an operation in the on-clause of a **give** statement, the programmer must ensure that the relevant pointer references memory holding the variable’s value. This can be either the address of a local variable or the address of memory allocated from the heap. Furthermore, the “len” field of a LenTerm character string variable must be set to the number of elements to be sent.

When a definite length LenTerm character string variable is given as the parameter to an operation in the on-clause of an **accept** statement, the ELROS generated code stores the characters received into the variable’s “val” array and sets the “len” field to the number received. If more than n characters are received, an exception is generated. If the variable is one of the NULTerm character string types, the ELROS generated code stores characters into the variable until either all of the received characters are stored, at which point the generated code terminates the string with a NUL character, or the number of characters exceeds the number given in the type declaration. In the latter case, the ELROS generated code raises an exception.

When the programmer uses an indefinite length character string variable as the parameter to an operation in the on-clause of an **accept** statement, the ELROS generated code allocates space from the heap to hold the value received. Consequently, any value in the variable’s pointer component is lost after the **accept** statement completes. If the variable is a NULTerm character string type, the generated code stores a NUL character at the end of the received value.

As described in section 3.4.1.5, the ELROS preprocessor generates routines that free space allocated either by the programmer or by the ELROS generated code that are referenced by the “val” field of a

DynamicArray. These routines are generated for each DynamicArray type as well as for all types containing one or more DynamicArrays.

The ELROS preprocessor also generates free routines for indefinite length string types and for other types that contain either DynamicArrays or indefinite length string types. These routines are accessed through the rofreei() and rofree() routines described in section 3.4.1.5.

For example, consider the following type defintion :

```
Record
    {
    gas_identifier element;
    Real(10,8) percentage;
    PrintableString(*) gas_name;
    } gas_percentage;

    DynamicArray gas_percentage GP_array[20];
```

Suppose within a program a variable of type GP_array is declared and intialized as follows :

```
GP_array gas_data;
            .
            .
            .
gas_data.len = 3;
gas_data.val = (gas_percentages *) malloc(sizeof(gas_percentages) * 3);

gas_data.val[0].gas_name = (char *) malloc(6);
strcpy(gas_data.val[0].gas_name, “Argon”);

gas_data.val[0].gas_name = (char *) malloc(5);
strcpy(gas_data.val[0].gas_name, “Neon”);

gas_data.val[0].gas_name = (char *) malloc(9);
strcpy(gas_data.val[0].gas_name, “Chlorine”);
```

At the point in the program where “gas_data” is no longer required, the programmer can write :

```
rofreei("GP_array", &gas_data, physics);
```

A free routine will attempt to free all DynamicArrays and indefinite length strings. However, there are situations when this can cause problems. For example, if the programmer has stored the address of a local variable in the pointer component of a data structure for which the preprocessor generates a free routine, the routine will attempt to return that space to the heap by calling “free()”. This will result in a system error that typically terminates the program.

Normally, ELROS generates a free routine for all types that are or contain DynamicArrays or indefinite length strings. However, the programmer can annotate data structures so that a free routine will not be generated. These annotations appear in angle brackets after the **Record**, **Union**, **Array** and **DynamicArray** reserved words. The reserved word **freeable** is used to instruct the preprocessor to generate a free routine and the reserved word **unfreeable** is used to instruct it to inhibit generation. Normally, if a data structure is unannotated, the preprocessor will generate a free routine. However, the programmer is allowed to annotate the interface definition statement to specify that all **Record**, **Union**, **Array** and **DynamicArray** types are **freeable** or **unfreeable**. The programmer can then override the interface default in the definition of particular types.

For example, suppose gas_data is initialized as follows :

```
gas_data.len = 3;
gas_data.val = (gas_percentages *) malloc(sizeof(gas_percentages) * 3);
gas_data.val[0].gas_name = “Argon”;
gas_data.val[1].gas_name = “Neon”;
gas_data.val[2]].gas_)name = “Chlorine”;
```

Calling rofreei() on gas_data will cause a system error, since the PrintableString components of each array element point to literal strings, not to memory allocated from the heap. Consequently, we could define GP_array as follows :

```
Record
    {
    gas_identifier element;
    Real(10,8) percentage;
    PrintableString(*) <unfreeable> gas_name;
    } gas_percentage;

    DynamicArray gas_percentage GP_array[20];
```

When defined in this way the call :

```
rofreei("GP_array", &gas_data, physics);
```

will not cause an error, since the PrintableString components have no free routine associated with them; thus, the free routine for GP_array will not attempt to free the gas_name component space.

V. Debugging ELROS Distributed Applications

Writing a distributed application is generally the easiest stage in its development. Getting a distributed application to work correctly is on average much more time consuming.

ELROS provides several tools that assist the applications programmer in debugging his code. These fall into three classes: 1) routines generated by the preprocessor to print variables declared as ELROS types, 2) a set of logging routines, and 3) a set of utilities for viewing and filtering the log data generated by these routines. The print and logging routines can be called from within an ELROS program or from a standard debugger such as dbx.

5.1 ELROS generated print routines

The ELROS preprocessor generates print routines for the basic ELROS types as well as for each type defined in an interface. These routines are accessed through one of four standard routines: 1) roprinti(), 2) roprint(), 3) froprinti(), and 4) froprint().

Roprinti() is called with the following parameters :

```
roprinti("type", &variable, interface)
```

The first argument is the ELROS type of the variable. The type is placed within double quotes so that the routine can be called from a debugger like dbx, which doesn't have ELROS types available in its symbol table. The second argument is the address of the variable to print. The third argument is the interface from which the type is drawn.

For example suppose we wish to print the value of the variable Gas_arg of type Gas from the physics interface. From an embedded language program the call would look like :

```
roprinti("Gas", &Gas_arg, physics);
```

Note that the interface is **not** surrounded by double quote marks. This is possible because the ELROS preprocessor declares a variable with the same name as the interface, which is included in its generated code and thus appears in the symbol table of the debugger.

To print the value of Gas_arg from the dbx debugger, the following dbx command is executed (the prompt dbx> is produced by dbx and not typed) :

```
dbx> call roprinti("Gas", &Gas_arg, physics)
```

The result of this call is :

```
= {
    [mixture] = {
                number = +3
                 component = {
                            [0] = {
                                  element = 1
                                  percentage = 0.5
                            {
                            [1] = {
                                  element = 12
                                  percentage = 0.4
                            {
                            [2] = {
                                  element = 7
                                  percentage = 0.1
                            {
                            [3] = {
                                  element = 0
                                  percentage = 0
                            {
                            [4] = {
                                  element = 0
                                  percentage = 0
                            {
                            .
                            .
                            .
```

```
                    [19] = {
                            element = 0
                            percentage = 0
                    {
            }
    }
}
```

The string "mixture" in square brackets at the beginning of the output text indicates which of the Union components is present in the variable Gas_arg. The remainder of the text is the value of this component. Notice that every component of the Array GP_array is printed, even though only 3 elements are valid. The print routine for the type "Gas" does not know that the value in "number" indicates which of the components in GP_array are valid.

The ELROS runtime system must know the interface from which a type is drawn in order to call the correct routine to print its values (e.g., two interfaces imported into the embedded language code might define types of the same name). Roprinti() specifies the interface name through its third argument. The ELROS runtime system remembers this interface name in an internal variable, allowing a subsequent call of roprint() :

```
roprint("type", &variable)
```

The first two arguments of roprint() are the same as roprinti(), the difference between the two routines being that the former does not specify an interface. Instead it uses the interface stored in the ELROS runtime system internal variable set by roprinti() (or froprinti() – see below). The use of roprint() relieves the programmer from typing the same interface name over and over again.

The output from both roprint() and roprinti() is directed to stdout. Sometimes it is desirable to print the value of an ELROS variable to a file. The routines froprinti() and froprint() are provided for this purpose. Froprinti() takes four arguments. The first argument is a file pointer to an open file into which the value of the variable is written. The last three arguments are the same as the three arguments of roprinti(). For example, the following code prints the value of the variable Gas_arg to the file "~nessett/dump" :

```
FILE *fd;

fd = fopen("~nessett/dump","w");

froprinti(fd, "Gas", &Gas_arg, physics);
                    .
                    .
                    .
```

The routine froprint() operates like roprint() except the value of the variable is printed to the file described by its first argument, rather than to stdout. Suppose there are two variables of type Gas, Gas_arg1 and Gas_arg2. The following code prints the values of these variables to the file "~nessett/dump":

```
FILE *fd;

fd = fopen("~nessett/dump","w");

froprinti(fd, "Gas", &Gas_arg1, physics);

froprint(fd, "Gas", &Gas_arg2);
                    .
                    .
                    .
```

5.2 Tracing ELROS program execution

In addition to routines to print ELROS data types, the ELROS package includes routines to trace the execution of ELROS embedded language statements. Four routines are provided : 1) start_log7i(), start_log7(), 3) stop_log7i(), and 4) stop_log7()[27].

When a programmer desires to generate a trace of the remote operations sent and received by a program, he calls start_log7i(). Normally start_log7i() is called from a debugger, although it may also be called from an embedded language program. The routine takes two parameters : 1) an active instance (i.e., one which has appeared in a bind statement) and 2) the name of a file to which trace information is output. The first parameter allows the programmer to specify a specific instance on which to conduct tracing. Tracing on that instance remains in effect until the instance is unbound or until stop_log7i() is called. If the first parameter to start_log7() is NULL (i.e., zero), global tracing is enabled. In that case activity on all instances is traced.

The second parameter to start_log7i() is the name of the file, enclosed in double quote marks, that will receive the tracing information. If start_log7i() is called with a NULL second argument, the following set of rules determines the trace output file. First, if start_log7i() has been called previously, the ELROS runtime system uses the file established by that call. Trace information is appended to it. Otherwise, an environmental variable, ELROSLOG7 is examined. If it contains a name, trace output is sent there. If ELROSLOG7 is NULL, a default name is used – ELROS.log7.

The routine start_log7(), taking only a filename enclosed in double quotes, turns on global logging. It uses the same filename conventions as start_log7i().

When a programmer wishes to turn off tracing, he calls stop_log7i(). Stop_log7i() takes a single parameter, an instance. Tracing for that instance is deactivated, while tracing for all other instances remains active. If the instance argument is NULL, global tracing is deactivated. The routine stop_log7() takes no parameter and turns off global tracing (although tracing on specific instances activated by calling start_log7i() remains in effect).

[27] There are also four other tracing routines, start_log6i, start_log6, stop_log6i and stop_log6. They generate tracing information of presentation-level activity (i.e., level 6 in the ISO OSI-RM hierarchy). They are not covered in the primer since they are of value mostly to system programmers.

To illustrate the use of these routines, suppose a programmer is debugging an ELROS embedded language program from dbx. The following command starts tracing on the instance "inst" with tracing information going to the file "~nessett/trace".

```
dbx>  call start_log7i(inst, "~nessett/trace")
```

The resulting trace information would appear as follows :

```
A00004ab4:    302   physics_client.c  inst
A00004ab4:          ENTERING give statement
A00004ab4:    305   physics_client.c  inst
A00004ab4:          GIVING compute_density(invoke_ID=12, linked_ID=0)
A00004ab4:              GAS= {
A00004ab4:                  [mixture] = {
A00004ab4:                      number = +3
A00004ab4:                      component = {
A00004ab4:                          0 = {
A00004ab4:                              element = 1
A00004ab4:                              percentage = 0.5
A00004ab4:                              }
A00004ab4:                          1]= {
A00004ab4:                              element = 12
A00004ab4:                              percentage = 0.4
A00004ab4:                              }
A00004ab4:                          2 = {
A00004ab4:                              element = 7
A00004ab4:                              percentage = 0.1
A00004ab4:                              }
A00004ab4:                          3 = {
A00004ab4:                              element = 0
A00004ab4:                              percentage = 0
A00004ab4:                              }
A00004ab4:                          4 = {
A00004ab4:                              element = 0
A00004ab4:                              percentage = 0
```

```
A00004ab4:                       }
                      .
                      .
                      .
A00004ab4:    312  physics_client.c  inst
A00004ab4:         EXITING give statement
```

The first field in every line is an integer constant that identifies the particular instance on which the activity occured. Each trace entry consists of multiple lines, the first of which indicates the line number in the source file where the trace entry was generated, followed by the source file name and the name of the instance variable used in the traced activity. Subsequent lines describe the activity in detail.

5.3 Using the ELROS tracing routines in realtime

Sometimes a programmer debugging an ELROS embedded language program desires to see the execution of ELROS statements in realtime as they occur. Two utilities are provided to support this style of debugging : 1) logfilter and 2) loganal.

Logfilter is a utility that accepts an ELROS trace file and filters out unwanted information. Its execute line is :

logfilter -a *filename @instance*

Filename is the name of a file containing ELROS runtime system trace information. If *filename* isn't specified, “stdin” is used. *Instance* is an integer constant that identifies an instance in a trace file (i.e., it is the integer constant that appears after the "A" on each line of the trace information). The output of logfilter is sent to “stdout”, so the programmer can view it on his terminal. If *instance* is not specified (i.e., the *@instance* is left off the execute line), all information in the trace file is sent to stdout with the instance information (i.e., the A00004ab4 which prefixes each line in the above example) stripped off.

Logfilter is useful when a complete file of trace information is available for analysis. To view trace information as it is generated requires the use of the second utility – loganal.

To use loganal, the programmer opens a second window and types :

loganal *filename* | logfilter -a @*instance* | more

Use of this execute line is predicated on the assumption that *filename* is the file established by the programmer when he called start_log7i() or start_log7(). Loganal opens up a pipe to *filename* and reads its contents as they are written. Each line read is sent to stdout, which in the execute line given above pipes it to logfilter. Logfilter discards all information unrelated to the instance *instance* and pipes the result to "more".

To examine all the tracing information generated by the ELROS runtime system, the programmer should use the following execute line :

loganal *filename* | logfilter -a| more

Appendix A - ELROS Summary

Below is a summary of the ELROS statements, support routines and execute lines covered in this document.

A.1 Interface Specification

interface *interface-name*;
type-definitions
operation-definitions
end;

The *type-definitions* and *operation-definitions* are collected into an interface called *interface-name*.

A.1.1 Type Definitions

Boolean, Integer, Real, BitString, OctetString, Enumerated, ObjectIdentifier, Null, NumericString, PrintableString, TeletexString, VideoString, VisibleString, IA5String, GraphicString, GeneralString

The basic types available to ELROS programmers.

Array *element_type name_of_array_type* [*number_of_elements*]

The Array type *name_of_array_type* is defined. By default it contains *number_of_elements* components, each of type *element_type*. The Array statement may only appear in ELROS interface specifications. When an Array variable is referenced in an operation, a runtime length may be attached within curly braces (i.e., {}). For an Array variable that is an **out** parameter, this value specifies how many elements to send. For an Array variable that is an **in** parameter, this must be a variable or some other lvalue. The variable or lvalue specifies the maximum length of the Array, which may be different from *number_of_elements* if the Array was allocated from the heap. It is set to the actual number of elements sent by the correspondent when the operation reference executes. ELROS Array definitions may only appear in interface specifications.

DynamicArray ***element_type name_of_array_type***

The DynamicArray type *name_of_array_type* is defined. Dynamicarrays expand to a C struct consisting of two components. The first is an unsigned long, “len”, that specifies the number of elements in the DynamicArray. The second, “val”, is an *element_type* pointer that holds the address of the DynamicArray data. A DynamicArray definition may only appear in ELROS interface specifications.

Record { ***record-list*** **}** ***record-type-name*****;**

The Record type *record-type-name* is defined. It contains components defined in *record-list*. Elements in *record-list* are semicolon-separated, each consisting of a type name followed by the name of the component or by a Record, Array or Union definition. A Record definition may only appear in ELROS interface specifications.

Union {***union-list*****}** ***union-type-name*****;**

The Union type *union-type-name* is defined. It contains components defined in *union-list*. Elements in *union-list* are semi-colon separated, each consisting of a type name followed by the name of the component. ELROS Unions expand to a C struct consisting of two components. The first is an integer, "tag", that identifies which union component is present. The second is a non-discriminated C union, "un", that holds the union component value. A Union definition may only appear in ELROS interface specifications.

names ***names-list;***

The definition of a set of names. The *names-list* consists of a comma separated list of triples, each consisting of the elements : 1) datatype, 2) identifier and 3) integer. Members of each triple are separated by white space. The datatype specifies the type of the name; the identifier is used to reference the name in subsequent operation parameter lists; and the integer is a unique numerical identifier of the name.

A.1.2 Operation Definitions

call ***operation_name*** ***operation_number***
in *in_parameters*,
out *out_parameters,*
inout *inout_parameters* ;

The definition of a **call** operation appearing in an interface definition. The operation is given an *operation_name* and *operation_number*. It has three classes of parameters : 1) *in_parameters*, 2) *out_parameters* and 3) *inout_parameters*.

invoke ***operation_name*** ***operation_number***
***parameter-list*;**

The definition of an **invoke** operation appearing in an interface definition. The operation is given an *operation_name* and *operation_number*. Its parameters are specified in *parameter-list*. Parameters are types defined previously in the interface (or one of the basic types) for positional parameters, or **names** defined previously in a *names-list* for named parameters.

result ***operation_name*** ***operation_number***
***parameter-list*;**

The definition of a **result** operation appearing in an interface definition. The operation is given an *operation_name* and *operation_number*. Its parameters are specified in *parameter-list*. Parameters are types defined previously in the interface (or one of the basic types) for positional parameters, or **names** defined previously in a *names-list* for named parameters.

error ***operation_name*** ***operation_number***
***parameter-list*;**

The definition of an **error** operation appearing in an interface definition. The operation is given an *operation_name* and *operation_number*. Its parameters are specified in *parameter-list*. Parameters are types defined previously in the interface (or one of the basic types) for positional parameters, or **names** defined previously in a *names-list* for named parameters.

A.2 Embedded language declarations

#include ***pathname***

The importation of the interfaces contained in the file *pathname*.

control(*interface name*) *instance_variable_list*;

The declaration of the instance variables in the comma separated *instance_variable_list*. Each instance variable is bound to the operations and data types defined in the interface *interface_name*.

bind_address *addr*;

The declaration of an address structure *addr*, which is an input parameter to **bind_addr** .

A.3 Instance control statements

init_puresocket_addr(&*addr*, *role*, "*TCP_address*");

The address structure *addr* is initialized. The *TCP_address* (a dotted IP address with an appended "+" and port number) loads the responder_address field of *addr*.

bind_addr(*instance*, *rules*, *&addr*, *timeout*);

The *instance* is bound according to the values in *addr*. If the role field of *addr* specifies initiator, the program attempts to contact the server at the address in the responder_address field of *addr*. If the role field of *addr* specifies responder, the program waits to be contacted at the address in the responder_address field of *addr*. The encoding rules used over the instance are specified in the *rules* parameter (currently, only asn_1_ber is supported). The *timeout* parameter specifies the number of seconds a program will wait for activity on the instance before taking a com_error exception.

unbind(*instance*);

The instance referenced by the variable *instance* is unbound from the correspondent.

A.4 Operation execution statements

***call_operation@instance(parameter_list)*;**

The operation *call_operation* is requested of the server accepting operations on *instance*. The *parameter_list* to this operation can be either positional or named (depending on the definition of the operation in the interface). In a positional parameter list arguments are separated by commas and their formal parameter cognates are identified by their position in the list. In a named parameter list arguments are identified by a name, which is separated with a colon from the argument's value.

accept [*operation-identifier*] *instance-variable*

{

on ***operation***$_1$***(parameter-list)***

C-statement$_1$

on ***operation***$_2$***(parameter-list)***

C-statement$_2$

.

.

.

on ***operation$_n$*** ***(parameter-list)***

C-statement$_n$

}

A set of operations, *operation$_1$*, *operation$_2$*,,..., *operation$_n$*, is associated respectively with a set of C language statements, *C-statement$_1$*, *C-statement$_2$*,..., *C-statement$_n$*. When a program executes the **accept** statement, it blocks waiting for the correspondent to request one of these operations over the instance identified in *instance-variable*. The arrival of *operation$_i$* loads the variables given in its *parameter-list*. Control then passes to *C-statement$_i$*. After execution of *C-statement$_i$*, control passes out of the **accept** statement.

give ***instance-variable***

C-statement

The *C-statement* is executed in the context of the instance identified by *instance-variable*. Within the *C-statement* embedded operation references identify operations transmitted over the instance. Each operation reference consists of an operation (which must be drawn from the interface to which the instance was associated in **bind_addr**) and its associated argument list. An operation identifier may be associated with the operation by including it in square brackets between the operation name and its parenthesized argument list.

exceptions

{

on ***condition$_1$***

C-statement$_1$

on ***condition$_2$***

C-statement$_2$

.

.

.

on ***condition$_n$***

C-statement$_n$

}

The **exceptions** clause can be associated with other ELROS statements including **bind_addr** , a **call** statement, an **accept** statement, a **give** statement and an **unbind** statement. The **exceptions** clause consists of a set of conditions, *condition$_1$*, *condition$_2$*,..., *condition$_n$* , associated respectively with a set of C language statements, C-statement$_1$, C-statement$_2$,..., C-statement$_n$. Valid conditions include com_error, internal_error, protocol_error, missing_op, missing_arg and bad_op_id. When the exception *condition$_i$* occurs, control passes to C-statement$_i$ in the **exceptions** clause. Once C-statement$_i$ executes control passes out of the **exceptions** clause. If an exception occurs for which no on-clause is provided, a default statement is executed that notifies the user that an error has occurred and then terminates the process.

A.5 Debugging statements

roprinti("*type*", &*variable*, *interface_name*)
roprint("*type*", &*variable*)
froprinti(*file_pointer*, "*type*", &*variable*, *interface_name*)
froprint(*file_pointer*, "*type*", &*variable*)

Routines for printing the value of ELROS types. Roprinti() takes three arguments : 1) a typename (*type*) enclosed in double quotes, 2) the address of a *variable*, and 3) an *interface_name*. The value of the variable is printed to stdout under the assumption its type is specified by the first argument drawn from the interface given in the third argument

Roprint takes two arguments: 1) a typename (*type*) enclosed in double quotes, 2) the address of a *variable*. It works exactly as roprinti(), except the interface is defaulted to the last referenced in a roprinti() or froprinti() call.

Both froprinti() and froprint() print variable values to a file, a pointer to an open file being their first parameter. Froprinti() takes three more parameters, which are identical to roprinti(), and it behaves like roprinti() except that values are output to a file rather than stdout. Froprint() takes two more parameters, which are identical to roprint(), and it behaves like roprint() except that values are output to a file rather than stdout.

start_log7i(*instance*, "*pathname*")
start_log7("*pathname*")
stop_log7i(*instance*)
stop_log7()

Routines for controlling tracing. Start_log7i() takes an *instance* and a file *pathname*. Tracing information for that *instance* is output to the file. If *instance* is null, global tracing is activated. If *pathname* is NULL, the environmental variable ELROSLOG7 is accessed for the filename. If ELROSLOG7 is NULL, the default filename, ELROS.log7, is used.

Start_log7() activates global tracing. Establishing the name of the tracing file is identical to that for start_log7i() if *pathname* is null.

Stop_log7i() de-activates tracing on *instance* (but not globally).

Stop_log7() de-activates global tracing (but not tracing activated calls to start_log7i()).

A.6 Debugging utilities

logfilter -a *filename* @*instance*

This utility searches the trace file *filename* looking for lines beginning with A*instance*. It then prints these lines to “stdout”.

loganal *filename*

This utility reads lines from *filename* as they are written by the ELROS runtime system and pipes them to “stdout”.

Index