ELROS – An Embedded Language for Remote Operations Service

M.L. Branstetter[a], J.A. Guse[a], and D.M. Nessett[a]

[a]Lawrence Livermore National Laboratory, Livermore, California 94550, United States of America

Abstract

We describe ELROS, an embedded language for programming distributed applications using remote operations (ISO 9072-1 and 9072-2). The motivation for this work is the development of both large scale scientific distributed applications as well as ISO application protocol implementations. We compare our work with other systems, such as ISODE and DAS, that support remote operations programming and point out certain advantages of ELROS. Finally, we suggest ways in which the current implementation of ELROS can be improved.

1. INTRODUCTION

Given the ubiquity of networking in modern computing, it is somewhat surprising that there are few distributed applications in production. By distributed application we do not mean networking applications that require constant human involvement, such as the use of file transfer facilities (e.g., FTAM, FTP), virtual terminal services (e.g., VT, TELNET), or electronic mail (e.g., X.400, SMTP). Rather, we mean a set of cooperating, communicating processes running on at least two different computer systems and carrying out a task without significant direction from a user.

We believe this paucity originates from several causes, including :

- Existing tools for building distributed applications are written for experts, not the average programmer. A typical application programmer must invest a significant amount of time learning the somewhat arcane interfaces currently used to write distributed applications. These interfaces include sockets or other communication libraries, various RPC or remote operation compilers, exotic communication and interface definition languages (e.g., ASN.1 [6] and the remote operation macros [8]), and the error handling facilities for these programming interfaces. In addition, most existing distributed application development tools rely on a stub generator, which forces the application programmer writing a server to use an unfamiliar, dispatcher-based programming style.
- The several available distributed application development environments enjoy some, but not all of the characteristics desirable for building production distributed applications. The SUN RPC/XDR tools [1] support high-performance communications for small data transfers, but are not optimized for large data exchanges (e.g., large arrays of reals) or highly structured data. In addition, these tools do not support asynchronous service requests, and require the programmer to build libraries of XDR encode/decode routines for data types other than the basic ones. The OSF RPC tools [2] are more con-

venient to use than the SUN RPC tools, but also do not support asynchronous service requests. OSF RPC/NDR uses the *receiver-makes-it-right* scheme of encoding data, adversely affecting interoperability by requiring each receiver to have the appropriate decode routines for every sender with which it communicates. The ISODE tools support both synchronous and asynchronous service requests, but require the programmer to learn ASN.1 and the remote operation macros. In addition, ISODE, being a reference rather than an optimized implementation, is not particularly well suited for efficiently moving large amounts of data between two correspondents. Other tools for developing distributed applications using ISO remote operations, such as DAS [12] also require the application programmer to learn ASN.1.

In full view of these problems, we set out to develop distributed application development tools that could cope with them or at least provide acceptable work-arounds. This paper describes one of these tools, ELROS[1], an acronym standing for Embedded Language for Remote Operations Service.

We had several objectives when designing ELROS, the most important of which are :

- Our funding sponsors are physical scientists, rather than members of the information processing industry. Consequently, we oriented our design heavily toward programmers who have little knowledge of distributed system technology. While ELROS supplies all the functionality required to write complete ISO application protocol implementations, such use requires somewhat more effort than writing scientific distributed applications.
- We believe good performance is a key to encouraging the development of distributed applications. This objective influenced several aspects of the ELROS design.
- A frustrating feature of other programming tools, especially stub generators, is the lack of control programmers have over the names of data structure elements, such as type identifiers, structure and union member identifiers, array identifiers and operation identifiers. An early and important design objective of ELROS was to give the programmer complete control over these names.
- In order to ensure the wide-spread availability of ELROS, we designed it so it could be easily ported to new systems. We modularized its runtime system so that it is relatively straight forward to port ELROS to new communications platforms (we currently support sockets [14] and the ISODE [13] platform). Our goal was a system that a moderately knowledgeable systems programmer could port to either a new systems architecture or a new communications platform in about 2 weeks.
- We chose the remote operations standard [8] and ASN.1 [6] as our initial protocol set. ELROS supports all 5 operation classes and all 3 as-

[1] "*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]

sociation classes and provides detailed control over invoke and linked ids. However, we designed ELROS so it could be modified to support other protocols, such as SUN RPC and OSF RPC, if our sponsors decide this is important.

- Experience in programming distributed applications suggested to us that a programming tool must provide significant support for debugging, so we made it a high priority. ELROS provides functionality that gives the programmer detailed control of timeouts from a debugger or from within program code, supports real-time and *a posteriori* activity tracing at both the remote operations as well as the presentation level, and automatically generates print routines for user defined data structures.
- Our use of other programming tools, such as ISODE, convinced us that the techniques they employ to pass error information between support libraries and an application program are cumbersome. Since ELROS is an embedded language, we decided to implement an exceptions facility that cleanly separates error handling from the other activity management code within a distributed application program.

2. THE LANGUAGE

We decided early to develop ELROS around the concept of an embedded language. We believed (and still believe) this approach makes distributed application programming more accessible to the average programmer. We reasoned that an operation used in different places in an application program can be placed within a procedure to conserve executable code space, in effect simulating an operation stub.

Detailed tutorials on the ELROS embedded language are found in [3] and [4]. Here we briefly describe its major features.

The embedded language is roughly divided into two parts : 1) the interface definition statements, and 2) the executable statements. Generally, interface definition statements appear only in an interface specification, which is usually a file of ELROS declarations maintained separately from invoker / performer (client / server) code. However, certain interface definition statements are used to declare ELROS variables within executable code.

2.1 Interfaces

ELROS interfaces are collections of type and operation definitions. Each interface is named, this name being used when the interface is imported into a section of executable code. Interfaces are imported textually, ELROS supports no run-time checks of interface compatibility (e.g., run-time checking of interface versions) other than a check of application and presentation context names, which may be supplied by the programmer.

All of the basic ASN.1 data types are supported by ELROS. Constructors for complex types such as arrays (both arrays with static as well as dynamic run-time lengths), records and unions are available. The programmer has complete control over the names used for all types, including record and union element names. A type resolution mechanism is provided that allows a programmer to select one of many identically named types or operations imported from different interfaces.

In addition to standard typing information, ELROS basic types take sizing information that indicates how they are to be mapped into native C types on the target machine. A configuration file accessible to the programmer gives him control over the mapping between ELROS basic types and native C types.

To support ease-of-use by programmers unfamiliar with the intricacies of ASN.1, data structure definitions are mapped by ELROS into default encodings. ELROS has an autotagging feature that ensures each element of a record or union has a unique ASN.1 type. This feature can be disabled by the programmer who desires to exercise control over the encoding of these elements.

ELROS provides a data structure annotation language that allows programmers to override the defaults, giving them complete control over how their data structures are encoded. Additionally, programmers can specify annotations for each element of a record to control when to send its value to a correspondent. These annotations indicate a particular element value that should not be sent (if the element is one of the basic ASN.1 types), the generation of a flag within the record to control when a given element should be sent and default values that a receiver should use to set the value of an element not received.

An ELROS interface specification also contains definitions of the operations it supports. All remote operation types are supported – invokes, results, errors, and rejects. Operations take either positional or named parameters. This flexibility is provided to ease the programming of scientific distributed applications. Operations taking named parameters are specified using names defined in the interface by a special ELROS statement. If an operation takes more than one argument or if it uses names, its parameters are encapsulated within a SET, which is sent as the single argument of the underlying remote operation. If an operation takes one positional parameter, it is not encapsulated within a SET, allowing the implementation of ISO standard application protocols.

2.2 Executable Statements

The basic construct in ELROS used to represent an active line of communications is the communications instance, which we abbreviate to the word "instance." There is an ELROS statement to bind an instance to a particular correspondent using its address, the exact structure of which depends on the communications platform selected by the programmer in the bind statement. ELROS also provides a way of binding an instance to either a memory buffer or a file. The programmer declares in the executable code whether an instance will be used to send/receive remote operations (control instances) or raw ASN.1 (data instances).

Applications can have more than one instance active (i.e., bound) at a time. Different active instances may concurrently use different communications platforms. Some concurrently active instances may be control instances, while others are data instances. With a few minor restrictions, instances may be passed as arguments to procedures and used concurrently by multiple threads.

The give statement asynchronously sends a series of operations to a correspondent over a control instance. Each operation referenced within a give statement is placed within a presentation data value (PDV) of an encapsulating presentation protocol data unit (PPDU). Arbitrary C code, including other ELROS statements, may be placed within a give statement, allowing looping, testing and (with restrictions) branching. Each time an operation reference is executed, a new PDV is placed in the PPDU. The PPDU is sent when control passes from the give statement.

The accept statement waits for one of a set of operations to arrive over an active control instance. The programmer specifies one or more operations that the program is willing to receive. The accept statement blocks until one of these operations arrives. Each candidate operation is specified in an on-clause[2] (similar to a label statement within a switch statement) that gives the operation name and the variables into which the operation arguments are stored. Within the operation on-clause may be placed arbitrary C code, including other ELROS statements. This allows the programmer to build operation contexts in an intuitively satisfying way in which the receipt of one operation enables the program to receive another operation from an encapsulated set (c.f. FTAM *regimes*).

ELROS also supports a synchronous call statement that sends an operation and blocks until a result is received. Call statements are identified separately from invoke, error, result and reject operations within an interface specification.

Operation references appearing within either a give or call statement may elide parameters, thereby optimizing the amount of data transmitted between invoker and performer. This feature is important when the programmer chooses to write a state-retaining performer (server) and the invoker (client) has sent identical argument values in a previous operation. The use of state-retaining servers is an important performance optimization technique when large amounts of data are transmitted between invoker and performer, which occurs in large distributed scientific calculations.

Data instances are used with the encode and decode statements. These statements allow the programmer to send raw ASN.1 to a correspondent without constraining its format by the remote operations standard. The encode statement operates very much like the give statement and the decode statement operates like the accept statement.

Data instances are also used to support the ASN.1 Any type. The data within an Any variable consists of the ASN.1 encoding of a specific type. To send an Any value, the programmer first binds an instance to the memory buffer associated with an Any variable and then uses the encode statement to place the appropriate ASN.1 into this buffer. The Any variable is then used like a normal ELROS argument. When an Any variable receives a value, the decode statement is used to move the ASN.1 placed within its associated buffer into a variable of the appropriate type. Special macros are provided to perform this processing atomically.

Most of the ELROS executable statements take an optional exceptions clause. Within the exceptions clause the programmer includes various on-

---

[2] The name "on-clause" derives from the use of the reserved word "on" followed by the name of an operation. This syntax is meant to suggest the idea, "on *receiving specific operation* do ...."

clauses that specify exceptional conditions that may occur on an instance. When one of these exceptions occurs, control passes out of the ELROS executable statement to the appropriate exceptions on-clause. Within the on-clause, the programmer may place arbitrary C code to handle the exception.

If the programmer fails to provide an exceptions clause for an ELROS statement, the ELROS preprocessor provides a default. If a programmer fails to supply all valid on-clauses within an exceptions clause, the preprocessor will provide default on-clauses for those that are missing.

We rejected support for user-defined exceptions and exception propagation, because we felt such a general approach would in effect result in an exception-handling mechanism design for the C language. We deemed such an approach to be impractical; first, because such an effort is already underway; second, because elaborate extensions to the C language might hinder rather than help the acceptance of ELROS in the user-community; and third, because the implementation cost would be unacceptably high.

Our exception mechanism does, however, depart radically from the de-facto philosophy of C in the following way. In C an exception is indicated through the calling interface by a return code. This return code is ignored by default, thus the default handler ignores the exception[3]. In ELROS the default handler aborts the entire program. We will be working with our sponsors to determine whether this decision is best or if ELROS default handlers should simply abort the instance and allow processing to continue.

ELROS also provides extensive debugging support. Print routines for each type defined within an interface are automatically generated by the preprocessor. The run-time system supports tracing of both layer 6 and layer 7 activity. Tracing may be started and stopped both by a debugger and by code contained within the program. Finally, a tool set to analyze layer 6 and layer 7 trace data, to display this trace data in real-time in an auxiliary workstation window and to show ASN.1 data in a context-free ASCII form are provided in the ELROS distribution.

3. THE PREPROCESSOR

The ELROS preprocessor accepts ANSI C programs with embedded ELROS constructs that have been previously run through the C preprocessor. It generates C code consisting primarily of calls to the ELROS run-time library. The output is enhanced with line directives so that subsequent compilation of its output by the C compiler will report the lines in the original source where ELROS syntax and semantic errors have occurred.

The parser first reads the contents of a configuration file that contains information specific to the target machine. We chose this strategy in order to separate configuration decisions from the core preprocessor code. Configuration information includes the endianness of the machine, the size of buffers for sending and receiving, and a mapping between different sizes of ELROS booleans, integers and reals and their corresponding C types.

[3]That is, the "handler" concept is implemented by an if-test and a code block, which if not present, represents the "default handler."

Interface specifications have their own grammar since they are primarily declarative in nature and naturally occur as a separately parsable unit. On the other hand, embedded code is substantially more difficult to parse since it occurs intermittently in C programs. ELROS statements can occur in C control structures and vice versa. Initially, we attempted to avoid parsing C but concluded that the parser would be cleaner and more flexible if all input was parsed in a uniform manner. Consequently, the preprocessor is driven by a suitably enhanced grammar of ANSI C. Nesting of ELROS statements is handled seamlessly by the parser as they are entered and exited. C code is echoed as it is encountered. The ELROS preprocessor also emits code for debugging and tracing the execution of an ELROS program.

4. THE RUN-TIME SYSTEM

The ELROS run-time system is a library supporting the code generated by the ELROS preprocessor. It consists of several major parts: communications platform dependent routines, protocol dependent routines, exception handling services, debugging and tracing facilities, and a multi-threading subsystem. The run-time system provides services corresponding to the presentation- and application-layer of the OSI Reference Model [10].

The ELROS run-time system incorporates a set of communications platform dependent routines that hide the details of the communications platform interface from the remainder of the run-time system. Through modularization of platform dependent code and orthogonalization of the low-level interface, ELROS can be ported to other communications subsystem interfaces with relatively little programming effort and expertise.

Our run-time system is a flat library; we try to avoid recursion wherever possible. We believe that using generated code in conjunction with primitive library routines for encoding and decoding yields as good or better performance than either a recursive descent or table-driven approach.

Our experience implementing the ELROS run-time system suggests that support of both definite and indefinite length encodings in the Basic Encoding Rules (BER) of ASN.1 [7] is unwise. We believe definite length encodings solve no problems, while creating at least two: 1) the sender must make two passes over data to compute the lengths and then encode them; and 2) the receiver must count bytes in order to use incoming definite lengths (a strategy which requires a stack in the general case).

In view of this analysis we based our implementation on indefinite lengths. We send indefinite lengths except for primitive types, and our receive code is most efficient when dealing with them. However, to ensure compatibility with other ASN.1 implementations, our receive code also processes definite length encodings. While we believe our approach to be both efficient and parsimonious, one disadvantage is a lack of compatibility with protocol definitions which require definite length encodings, such as SNMP [5] and the distinguished encoding rules of X.509 [11]. We believe the developers of SNMP were inadequately familiar with the issues surrounding the length encoding problem, since our experience shows that use of indefinite lengths is far simpler and much more efficient than the use of definite lengths.

Exceptions in an ELROS program can originate either in the run-time system or in the code generated by the preprocessor. Both varieties are handled by the run-time system, which associates each exception with an instance, solving a number of problems. It provides a natural place to store state regarding the exception. That a particular instance is being serviced when the exception is raised, means that there is an enclosing ELROS language construct that can cleanly handle the exception; thus, we did not need to alter the semantics of any C statement to accommodate our exception mechanism. It also means that our termination model semantics have a clearly defined boundary (the scope of the enclosing ELROS statement), which leads to simple recovery strategies (generally aborting the instance).

Finally, the ELROS run-time system uses the PADS multi-threading support subsystem, a library developed at Lawrence Livermore National Laboratory to provide machine-independent threads. Because of the relative ease of implementing a PADS interface over other multi-threading implementation, ELROS is easily ported to commercially available thread packages. It is our intent to convert the ELROS run-time system to the POSIX 1003.4a multi-threading standard when it becomes widely available.

5. RELATED WORK

The ELROS program development system provides functionality similar to other public domain systems, including the ISO Development Environment [13] and the Distributed Application Support Package [12]. While other tools for programming distributed applications exist, we limit our discussion to public domain systems supporting the ISO standards.

5.1 ISODE

The ISO Development Environment (ISODE) is a free and widely-available implementation of remote operations and ASN.1 [13]. In early versions, the ROSY-POSY-PEPY tools were the basis for ISODE layer 6 and 7 programming capability. In the current release, these are replaced by the ROSY-PEPSY tool set.

Depending on the release, interactive use of the ISODE programming tools requires separate invocation of either two or three compilers (in addition to the C compiler), each of which generates a plethora of files. The application programmer must be concerned with the existence and naming conventions of these files to correctly pass them between the compiler tools and **cc**.

ELROS takes a different approach, its preprocessing and compiling steps being managed by a driver (called **ecc**) that manipulates all intermediate files. This convention is modeled on the **cc** driver of UNIX™, which makes it natural for the typical application programmer.

The ISODE programming tools are stub generators, which are not integrated into the C host-language. To use remote operations, the programmer must call library routines, which makes remote operation calls significantly different from local procedure calls. We regard this as a significant barrier to the acceptance of distributed programming environments by the general programmer population.

ELROS extends the C language to support syntax for using remote operations. To a great extent, these extensions look very much like normal C function calls.

The ISODE programming tools generate subroutines for each data type used in an interface. The advantage of this technique is that it reduces code space. Its disadvantage is it increases the overhead of manipulating complex data structures.

ELROS generates code to manipulate data structures inline, resulting in more complex, but faster code. A potential disadvantage of the ELROS approach is that if operations appear often textually, the size of the object code can increase. We believe this effect should be minimal, since the proportion of code dedicated to "remote communication" in a typical application is low, making even a marked increase in the size of this part of the application object code insignificant with respect to overall application size. Furthermore, if certain operations are used extensively, the programmer can place them in C procedures, thereby reducing the code generated.

5.2 DAS

Another public domain implementation of remote operations is the Distributed Application Support Package (DAS) developed at the University of British Columbia [12]. DAS is a set of library routines and an ASN.1 compiler for generating encode/decode routines. It provides an independent implementation of the ISO upper layer protocols and enjoys several advantages over ISODE. First, it internally supports multi-threading. While ELROS operating over the ISODE communications platform also supports multi-threading, we had to make some compromises to achieve this (e.g., the ELROS bind statement blocks the process briefly because of the way ISODE is implemented – a work around is currently being investigated). Secondly, the DAS ASN.1 compiler produces more efficient encode/decode routines than the presentation element approach of ISODE.

ELROS, on the other hand, is easier to use than DAS, which is based on a library call approach to remote operations. Furthermore, ELROS supports all five operation classes, while DAS only supports classes one and two. We have not yet had the chance to compare the performance of ELROS and DAS, so we cannot say which is more efficient.

6. CURRENT STATUS, PERFORMANCE, AND FUTURE WORK

6.1 Current Status

ELROS has been ported to most varieties of SUN workstations, CRAYs using the UNICOS operating system and Vax systems running Ultrix. It is scheduled for beta release in early 1992.

We have developed a prototype preprocessor, which is complete, to test the run-time system. The prototype preprocessor compiles a subset of the ELROS embedded language and has been used to build several simple clients and servers that interact with each other as well as with some of the servers supplied with ISODE. The prototype preprocessor also was used to investigate ELROS performance.

6.2 Performance

We compared ELROS, ISODE (v6.8) and XDR performance to determine whether our intuition about the superior performance of inline generated code was correct. We selected the transmission of an array of 30,000 elements from an invoker to a performer and then back again as representative of what a typical application of our customers might require.

The following table illustrates average wall clock times (in seconds) to send and receive three different data types using different communications subsystems. The test involved sending a 30,000-element array in both directions between two Sun-3 workstations connected by an Ethernet. The array consisted of either integers (**int**), reals (**double**) or character strings of 8 characters. The first test program is written entirely in ISODE using the most efficient means available.[4] The second and third are written using ELROS, where the second uses the ISODE communications platform (session-layer, except for presentation-layer binding) and the third uses the socket platform. The fourth test program is written using sockets and XDR encode/decode routines.

	Integer	Real	String[8]
ISODE (v6.8) Rosy/Posy/Pepy	**47**	**397**	**124**
ELROS (over ISODE)	**23**	**67**	**26**
ELROS (over sockets and TCP/IP)	**13**	**43**	**13**
XDR (over sockets and TCP/IP)	**3**	**5**	**18**

Table 1 – A comparison of communication subsystem performance
(time in seconds to echo a 30,000 element array)

A detailed discussion of these results, with the exception of the XDR tests, is found in [9]. We summarize the analysis here. Some of the reasons for the better performance of ELROS with respect to ISODE include the cumbersome data structures created by ISODE to encode and decode data, the lack of an incremental SSDU manipulation capability in ISODE, the amount of real arithmetic done by the ISODE real number routines,[5] the intricate data structures used to represent strings in ISODE, and the fact that the ISO protocols are more complicated than the Internetwork protocols.

XDR significantly outperformed ELROS in some cases. This is primarily because the XDR encoding for integers and reals is exactly like the native

[4] We didn't test the performance of ISODE using Pepsy, since it represents SET OF and SEQUENCE OF only as linked-lists, a strategy which is far less efficient than using a contiguous array. Since ELROS uses contiguous arrays, we felt it would be unfair to compare it to Pepsy.

[5] ELROS not only incorporates the ISODE encode/decode algorithms (which are very portable), but selectively uses the representation of certain native formats (such as IEEE floating point) to improve efficiency.

format of the Sun-3, and because XDR has no tagging and length information. We intend to investigate strategies for enhancing ASN.1 transfer syntax performance and believe there are ways to match or exceed XDR performance with suitable presentation-level optimizations.

6.3 Future Work

ELROS has a number of deficiencies as it is currently implemented. We discuss these here and point out how we expect to correct them in the future.

Each instance is associated with a particular interface when it is declared and so it is of a different type than an instance associated with another interface. Consequently, communication instances are conceptually C typed variables, but our current implementation doesn't treat them as such. For example, instances cannot be members of C constructed types, such as structs, arrays or unions. Also, expressions yielding a valid instance may not appear where an *instance-variable* is expected. We hope to enhance the preprocessor to remove these artificial restrictions.

Interfaces cannot incorporate other interfaces to provide upward compatibility or to reuse declarations, although a similar effect can be achieved using nested include statements. We believe that the most natural way to share information between interfaces is through an inheritance mechanism. Such a mechanism would provide a general and structured approach to interface maintenance and development.

The goal of the ELROS embedded language is to free the applications programmer from understanding irrelevant details of the underlying communications subsystem. One area where this is not yet possible is the requirement that the programmer provide a communications address when binding to a correspondent. Currently, we allow the programmer to initialize a platform-specific address and use that address in the platform-independent bind statement. We would like to integrate the bind semantics with a suitable naming service, such as X.500, so application programmers and users of distributed applications need only deal with intelligible names.

Our current implementation of the bind statement does not allow the programmer to provide a User Data parameter. This is a deficiency we intend to correct as soon as the current release has stabilized.

Our implementation of ELROS statement blocks is problematic with respect to goto, return, break, and continue statements. Currently, the preprocessor generates diagnostic messages when the use of these statements would interfere with the generated code or cause unpredictable results. However, we believe that ELROS statements should integrate seamlessly and intuitively into standard C and are investigating ways to correct this problem.

During our design of ELROS, we included statements to send/receive unstructured bulk data. We envisioned support of protocols such as RTSE through these statements. Due to resource restrictions, we chose not to implement this functionality in the first release of ELROS. While we recognize the importance of a reliable bulk data transfer protocol, we did not feel it represented an important initial customer requirement. However, we plan to support this part of the ELROS embedded language in a future release.

An important feature of the remote operations protocol is its ability to support at-least-once, at-most-once, and exactly-once operation semantics. Currently, these semantics must be implemented by the application programmer. We hope to extend the call statement to allow a reliability annotation so that application programmers can specify the semantics they desire from the run-time system.

Finally, the ELROS preprocessor generates ASN.1 header information used by the run-time system. This information is copied from constant tables when an application sends encoded data. However, the run-time system of the receiver decodes all headers as they arrive, making no use of these constant tables. We believe that one useful approach to improve performance is not to decode the received headers, but to merely compare the leading byte or two with predetermined values, and branch based on these comparisons. Since the majority of the information in the header is redundant and unnecessary, we believe such an approach would yield improved performance.

7. ACKNOWLEDGMENTS

A number of people have worked on the ELROS project, helping with both its design and implementation. Gary Long and Ashley Allen contributed to some of the early ELROS design work. Robyne Sumpter worked on parts of the run-time system and Rob Shaw converted the preprocessor parser to an ANSI C grammar. Finally, Linda Stanberry has recently joined the project, porting ELROS to a number of different systems, and we expect her to make major contributions to ELROS in the future.

"Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48."

8. REFERENCES

1. Network Programming. Sun Microsystems, part no. 800-1779-05, Sun Microsystems Inc., Mountain View, CA.

2. OSF Distributed Computing Environment, Release 1.0 Developer's Kit Documentation Set (Preliminary), February 5, 1991. Open Software Foundation, 11 Cambridge Center, Cambridge, MA.

3. M.L. Branstetter, J.A. Guse and D.M. Nessett, "An ELROS Primer," Lawrence Livermore Nat. Lab., Livermore, CA, Rep. UCRL-MA-108543, October 30, 1991.

4. M.L. Branstetter, J.A. Guse and D.M. Nessett, "Using ELROS To Implement ISO Application Protocols," Lawrence Livermore Nat. Lab., Livermore, CA, Rep. UCRL-MA-108542, October 30, 1991.

5. Jeffrey D. Case, Mark S. Fedor, Martin L. Schoffstall, and James R. Davin, "A Simple Network Management Protocol," Request for Comments 1098, DDN Network Information Center, April, 1989.

6. CCITT, "Specification of abstract syntax notation one (ASN.1)." International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.208.

7. CCITT, "Specification of basic encoding rules for abstract syntax notation one (ASN.1)." International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.209.

8. CCITT, "Remote Operations: Model, Notation and Service Definition." International Telegraph and Telephone Consultative Committee, 1988. Recommendation X.219.

9. James A. Guse, "ELROS Performance," *in preparation*.

10. ISO,"Information Processing Systems – Open Systems Interconnection – Basic Reference Model. International Organization for Standardization and International Electrotechnical Committee, 1984." International Standard 7498-1.

11. ISO,"Information Processing Systems – Open Systems Interconnection – The Directory – Authentication Framework. International Organization for Standardization and International Electrotechnical Committee, December, 1988." International Standard 9594-8.

12. G. Neufeld and M. Goldberg "DAS: An OSI Distributed Application Service", IFIP WG 6.5 International Symposium on Message Handling Systems and Application Layer Communication Protocols, Zurich, Oct 1990.

13. Marshall T. Rose, Julian P. Onions and Colin J. Robbins, "The ISO Development Environment: User Manual (version 7)," July 19, 1991.

14. Stuart Sechrest, "An Introductory 4.3BSD Interprocess Communication Tutorial," Unix Programmer's Manual Supplementary Documents I, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, pp. PS1:7-1 – PS1:7-25 (available from the USENIX Association).