An Object-Oriented Re-Engineering of the Remote Procedure Call, STREAMS and Transport Layer Interface.

C. Enright enright@dmi.usherb.ca
and M. Barbeau barbeau@dmi.usherb.ca
Unversité de Sherbrooke
Sherbrooke, Qc, J1K 2R1

***Abstract* - The capacity to use object-oriented features when using operating system primitives such as Remote Procedure Calls (RPC), Streams and Transport Layer Interface (TLI), is extremely limited. These operating system primitives were constructed to be used within a structured programming paradigm. With the advent and proliferation of object-oriented technology the use of these primitives in their present state within this paradigm is difficult. This work describes an intermediate step between a fully object-oriented implementation of these primitives and their present form. This intermediate state allows for the use of RPCs, STREAMS, and TLI within an object-oriented paradigm using all the benefits therein, such as inheritance and polymorphism.**

keywords: UNIX System V, RPC, TLI, Streams, object-oriented, C++.

I. INTRODUCTION

During the past several years, there has been a substantial increase in use of the object-oriented paradigm within the world of software engineering. This increased use has taken place at virtually every phase of the software engineering life-cycle, from *analysis of needs* right through to *implementation and testing*. Although this proliferation of the object-oriented paradigm is easily observed both in the commercial and scientific literature, many of the basic application building primitives available to system programmers have not taken this step into this brave new world.

The impetus for this work was derived from a project that specified an object-oriented re-write of the OSI transport layer carried out at the Université de Sherbrooke. The project was implemented using STREAMS and TLI, even though these primitives were not designed or implemented in an object-oriented manner. The difficulties encountered due to the structured, non-object-oriented nature of the TLI and STREAMS code lead to the work described in this article.

The contribution of this work is to offer an outline for an intermediary state between the structured implementation of these primitives and an *object-wrapped* version allowing access to object-oriented features such as, inheritance, function overloading, aggregation, genericity encapsulation and polymorphism to system programmers using these application building operating system primitives. Specifically, Remote Procedure Call (RPC), STREAMS, and Transport Layer Interface (TLI) as implemented in UNIX system V, are excellent candidates for this procedure.

This paper is composed of six sections: the first is the introduction, the second is a literature survey of object-oriented telecommunications, the third describes the UNIX V operating system and the component primitives being studied, the fourth outlines the object-wrapping technology and the fifth presents an example, based upon TLI, of the code used to *wrap* the existing primitives within a set of C++ classes, and the final section presents the conclusions of this work.

II. LITERATURE SURVEY OF OBJECT-ORIENTED TELECOMMUNICATIONS

The basic tenants, concepts and structures of the object-oriented technology are well defined, stable and starting to appear in high performance design systems [3]. Integration of the technology into the existing software and organistic systems has proven both possible and effective [14]. With this evidence, object-oriented software development has reached a level of critical mass and should prove to be the 90s analog to the 70s advance in software development through structured programming. In this section, several articles related to the present work are discussed.

An object-oriented model of the OSI layers using a parallel, object-oriented specification language called Mondel was done by Mondain-Monval and Bochmann [12].

A second experiment was carried out at Bellcore [2] using *object* wrapping technology. This technology outlines a strategy for *wrapping* the non-object-oriented software within an object-oriented envelope. Alfano and Mathews used this approach successfully. Alfano and Mathews stated initially that the developers thought that a prototype was a complete waste of time and that they should have made the jump directly from paper to production. However, a year latter the developers' who had initially thought the prototyping stage to be a waste of time, now considered it, in retrospect, the single most important and valuable design activity.

A third project undertaken by Feldhoffer [6] described an object-oriented modeling of an OSI basic reference model compliant application layer. This work was based on the modeling of the application layer concepts, such as Application Service Elements (ASE) using an object-oriented approach. Several refinements and enhancements of ASE properties were achieved by using the object-oriented paradigm

A fourth project performed by Koivisto and Reilly [8] concentrated on how aspects of object-oriented development can use existing ASN.1 (Abstract Syntax Language 1) code. The primary focus of this work was to create supporting code for object-oriented telecommunications software in C++ from ASN.1 code artifacts. This object-oriented code was integrated into an existing functional telecommunications protocol, called OTSO, and an experimental tool called CLASN was developed from an existing C base ASN.1 compiler.

A fifth project done by Guenther and Wackerbarth [7] approached the concept of the object-oriented paradigm for large scale software projects. The principal effort of this study was to verify that object-oriented development could be used for large scale software projects. The conclusions drawn from this work are in favor of object-oriented development for switching software. The development was simplified and stabilized, even though the software developers had to follow new paths of development. The one negative aspect was the decreased performance of the switch, thought to be a result of inheritance.

In a sixth project, Liu [11], presents an object-based approach for protocol implementations. Each state in an FSM is implemented as an object. The International Standards Organization (ISO) Open Systems Interconnection (ISO) Transport Protocol 0 (TP0) example is given. The member functions of objects are based on event triggers and the sub-states of said objects are represented the by sub-classes in question. Incremental development is proposed and reuse of constructs is suggested.

Object-oriented implementation of telecommunications software is also discussed by Divin and Petitpierre [5], their work is based on a new design of C++ specifically for protocols. Further references to object modeling for telecommunications software can be found in research papers by Abe [1], Mann [10] and Kadoch, Erradey and Bochmann [9].

III. THE UNIX V OS AND ITS TELECOMMUNICATIONS PRIMITIVES: RPC, STREAMS, AND TLI

UNIX system V release 4 (SRV4) is a unique version of the standard multi-user, multi-tasking operating system (OS). Its uniqueness, in part, is based on its support for multiple standards (such as, System V, SunOS, BSD, Xenix, SCO) within one OS. This multiplicity of standards support allows access to many different OS primitives, structures and functions at the networking level [13]. This study will focus on three of these primitives, Remote Procedure Calls (RPC), STREAMS and Transport Layer Interface (TLI).

RPCs are of interest since they allow system programmers to have access to procedures running in separate processes on local or remote machines through a telecommunications network. This is done via a programming interface that does not require the programmer to manage the details of the communication while making use of its facilities. For example, consider a system of distributed databases, the *sorting* procedure could be located in one machine on the network and all the separate database managers could use an RPC to call the said sorting procedure. The application, using an RPC, can use any transport provider available, whether the provider is connection-oriented or connectionless (of course the choice affects the type of remote procedure that can be designed).

STREAMS (a subsystem, not the return of *fopen*), furnishes a framework upon which communications services can be constructed. These services can be between process within the same memory space, or between processes running on different computers, furthermore, they could be between terminals and host computers. The STREAMS interface is placed at a much lower level than that of the RPC. STREAMS in their simplest incarnation supply a bi-directional data pipe between a user level process and a kernel level device driver. Data flows *downstream* (from user to driver) and *upstream* (from driver to user) by way of messages, although if necessary, the STREAM elements can treat the data as a byte stream. As an example, consider an OS in which multiple processes wish to write to a printer deamon implemented as a device driver, each process could then use a STREAM type interface.

TLI is usually the principal networking interfaced used in SRV4 programming. TLI provides an abstract view of a network based on layer four (transport) from the OSI seven layer model. This interface operates end-to-end, allowing applications to establish communication even though they have no knowledge of the networks topology. TLI can either be connection-oriented, using TCP, or connectionless using UDP. Furthermore, the TLI services conform to many other standard protocol suites such as, TCP, XNS, SNA and ISO. There are three components that form the TLI, first a user level library (provides interfaces to the transport layer for the application), second, a transport provider (TP) (furnishes transport and could be implemented by STREAMS) and third, *TMOD*, is a module that maps TLI primitives to TP primitives and is based on STREAMS.

In the remaining sections of this article the object-oriented wrapping will be defined and the object-wrapping - re-engineering of TLI will be presented. The TLI primitive was chosen since it is the primary networking interface used in SRV4; moreover, it provides all the necessary structures to describe object-wrapping within the context of this article and due to spatial constraints all three primitives could not be presented.

IV. OBJECT-WRAPPING TECHNOLOGY

The object-wrapping technology has been used successfully in various experiments, as described in section two. Essentially, this technology takes structured, non-object-oriented code and wraps it in an object-oriented envelop. Most code that was well written in a structured fashion (the modules should have high cohesion and low coupling) can be taken and re-engineered in an object-oriented manner. But, there are some circumstances where structured code should not be object-oriented re-engineered using wrapping. For example, when the existing code can not be grouped into a coherent class structure, that is, the code does not form a representative object within the problem domain; or under conditions of poor structured code that has either low cohesion properties or high coupling properties or both.

The object-wrapping concept involves looking at the problem domain and verifying that the coded functions can be grouped in a framework that can be interpreted as a coherent group within the problem space. This group of functions provides the basis for the definition of classes and their object instances. Once the functions have been coherently grouped they usually become the methods for the relevant class.

This method has been used to object re-engineer the TLI, as described in the next section.

V. TRANSPORT LAYER OBJECT INTERFACE (TLOI) (THE OBJECT WRAPPING OF TLI)

TLI has three major duties. The first is connection establishment and release. The second is data transfer. The third is error handling. Connection establishment can be further subdivided into the building of an address, and the opening and closing of the connection. These duties should provide a suitable starting point for the class formation of TLOI. Figure one illustrates a mapping of TLI functions to TLOI methods and eventually the classes.

Due to this strong delineation of structure inherently present in the TLI, the derivation of the class structure was straightforward. The *TLOI* class is composed of three classes. First, the *connection establisher/releaser* class whose responsibilities are to handle connection establishment and release, with all the necessary details. This class is itself composed of two classes, the *address_manager* class and the *connection_manager* class, which perform the duties of building the machine address of the desired destination and open and closing the connection to said machine respectively. The second class, *data_transfer,* is responsible for the details of transferring the data between the applications. Last, the *error_handler* class is charged with decoding trapped errors and displaying relevant messages to the user.

Figure two is a Booch notation [4] (a common object-oriented analysis and specification notation) representation of the TLOI class structure; dotted clouds represent the classes. Clouds within clouds are the aggregated classes forming the outer class. Each class has its methods described in the text

TLI functions

used for connection establishment/Release

address management

getnetconfigent()
freenetconfigent()
netdir_getbyname()
netdir_free()

connection management

t_open()
t_bind()
t_close()

used for UDP Data Transfer

t_senudata()
t_rcvudata()

used for TCP Data Transfer

t_connect()
t_snd()
t_rcv()

used for error handling

t_rcvuderr()
t_errorno()
t_error()
netdir_perror()

Fig 1 Mapping of TLI functions to TLOI classes-methods.

under the class name. This class framework is an abstract object-oriented representation of the new TLOI. The existing functions used to perform each of the TLI procedures will

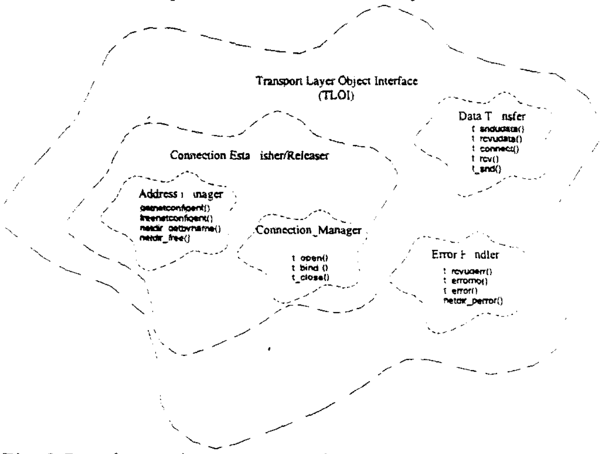

Fig. 2 Booch notation representation of TLOI class structure.

become the methods of the newly formed classes. For example, the *getnetconfigent* function will become a method of the *address_manager* class. All the functions of the TLI are treated in the same way; being allocated to their respective classes. Figure three provides an example of the C++ code used to implement this class hierarchy. This code is only partial in nature, that is, no data structures are included; moreover, it is meant as an outline to help the reader understand object-wrapping and does not conatin the complete code necessary to implement the TLOI.

Both RPC and STREAMS can be treated in a similar manner achieving object-wrapped results.

```
class TransportLayerObjectInterfaceTLOI {
public:
        class connection establishmer_releaser {
                public:
                        class address_manager {
                                public:
                                        getnetconfigent();
                                        freenetconfigent();
                                        netdir_getbyname();
                                        netdir_free();
                                private:...};
                        class connection_managemer {
                                public:
                                        t_open();
                                        t_bind();
                                        t_close();
                                private:...};
                private:...};
        class data_transfer {
                public:
                // UDP //
                        t_senudata();
                        t_rcvudata();
                // TCP //
                        t_connect();
                        t_snd();
                        t_rcv();
                private:...};
        class error_handler
                public:
                        t_rcvuderr();
                        t_erromo();
                        t_error();
                        netdir_perror();
                private:...};                   }
```

Fig. 3 Partial listing of C++ code for TLOI Class.

VI. CONCLUSIONS

In effect, we have taken a non-object-oriented set of UNIX system V primitives (RPC, STREAMS, and TLI) and through a process of *object-wrapping* we have brought the object-oriented attributes to them. These newly created object-oriented primitives behave as if they were originally written within the object paradigm. All the positive attributes of the object-oriented paradigm are available from these operating system functions. The new procedures, Remote Object Call (ROC), Object Streams and Transport Layer Object Interface (TLOI), allow system programmers object-oriented interfaces when developing and programming system applications bypassing the old standby structured routines.

This *wrapping* technology provides an efficient intermediary way to utilize object-oriented facilities without a complete re-write of the structured code. It is useful until such time when natively object-oriented primitives are mature and available. Object-wrapping technology does not work for all situations, as was discussed in section three, and it is not a substitute for a complete and total re-write, and re-design of the primitives in question. Within the context of this study the *wrapping* proved to be effective and useful.

For future study we propose the iteration of an increasingly more detailed and refined object-wrapping. This approach would see the wrappings descending further into the structured code on every iteration. This iterative-wrapping approach could be used to negotiate the transition from legacy structured code to fully object-oriented code.

VII. REFERENCES

[1] T. Abe, Y. Mitsunga, et al., "Application of Object-Oriented Techniques to Subscriber Cable Networks",*Globecom, Orlando, December*, 1992 pp. 260-264.

[2] J. Alfano and P. Mathews "Introducing object-orientation in a main frame software development organization",*Globecom, Orlando, December, 1992.* pp. 274-278.

[3] T. Biggerstaff, "Design recovery for maintenance reuse", IEEE Computer, Vol. 22, No. 8 (July 1989), pp.36-49.

[4] G. Booch, *Object Oriented Design and Analysis with Applications*, Benjamin Cummings, Redwood, Cal.., 1994.

[5] A. Divin and C. Petitpeirre, "An object-oriented method for implementing protocol stacks", *Proc.of FORTE*,May 14, 1993.

[6] M. Feldhoffer, "Object-oriented modelling of the application layer structure", *ULPAA*, Neufeld & Plattner (Eds), Elsevier Science Publishers B.V., North-Holland, IFIP, 1992.

[7] W. Guenther, Wackerbarth, "Object-Oriented Design of ISDN Call-Processing Software", IEEE Communications, 04 1993.

[8] J. Koivisto and J. Reilly, "Generating Object-Oriented Telecommunications Software Using ASN.1 Descriptions", ULPAA, Elsevier Science B.V. NorthHolland, 1992, IFIP.

[9] M. Kadoch, S. Erradi and G.v. Bochmann, *Object-Oriented Methodology of the MHS Protocol*, École de Technologie Supérieure, Technical Report, Dec. 1993.

[10] S. K. K. Man, "Experiences in using object-oriented technology in the development of the SPACE system",*Globecom, Orlando, December*, 1992, pp. 265-272.

[11] C. Liu, "An Object-Based Approach to Protocol Software Implemenation", Computer Communications Review of the ACM, proceedings of SIGCOMM, 1994,pp. 307-316.

[12] P. Mondain-Monval and G. v. Bochmann, *Object-oriented Software Architecture for the OSI Basic Reference Model.* Département d'informatique et de recherche operationelle. Université de Montréal, 1990.

[13] Rago, *UNIX System v Networkin Programming*, Addison-Wesley Publishing, Don Mills, 1993.

[14] E. Yourdon, *Object-Oriented Design: An Integrated Approach*, Prentice-Hall, Toronto, 1994.