Implementing Communication Protocols Using Object-Oriented Techniques

R. Greg Lavender, Dennis G. Kafura, and Chris J. Tomlinson

TR 92-11

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

April 13, 1992

Implementing Communication Protocols Using Object-Oriented Techniques

R. GREG LAVENDER

MCC
3500 W. Balcones Center Dr.
Austin, TX 78759-6509
512.338.3252
lavender@mcc.com

DENNIS G. KAFURA

Department of Computer Science
562 McBryde Hall, Virginia Tech
Blacksburg, VA 24061-0106
703.231.5568
kafura@cs.vt.edu

CHRIS J. TOMLINSON

MCC
3500 W. Balcones Center Dr.
Austin, TX 78759-6509
512.338.3765
tomlinson@mcc.com

Paper Category: Experience

Abstract

Appropriate use of object-oriented programming mechanisms in the implementation of communication architectures results in a type structure that reflects the architectural model and increases the run-time performance of the protocol machines at each layer. This result is important since it contradicts the assertion by some that layered protocol architectures, such as the ISO Reference Model, necessarily suffer in performance because of layering, thus forcing implementors to violate the layering principle in search of efficiency. The result substantiates the experience of others, notably Clark, that sound network programming methodology leads to efficient implementations that are also understandable. In this paper, elements of an object-oriented implementation of the upper layer OSI protocols are presented. Our goal in the paper is two-fold. First, to communicate to software engineers, particularly those developing systems software, that object-oriented programing techniques facilitate and enhance the implementation of layered architectures. Second, to convey to language designers our experience using object-oriented language features in building communications protocols. We focus on the requirements for the upper layer OSI protocols and illustrate how inheritance, subtyping, and polymorphic functions effect their implementation.

1. Introduction

An object-oriented approach to developing layered protocol architectures has useful implications for object-oriented applications. Layered protocol architectures, such as the ISO Reference Model for Open System Interconnection (OSI) [CCITT 1989], are commonly interpreted as monolithic, *horizontally* composed structures. Protocol implementations are sometimes structured to reflect the layered service model which they implement. An argument can be made that this structure is a poor engineering choice, for performance reasons, when a layer is equated with a process or when the layer interface requires copying and buffering of data passing through each layer. Some might even say that this approach is inappropriate since the layered model reflects ISO committee decomposition rather than functional decomposition.

Strict horizontal composition often dictates intra-layer multiplexing and inter-layer communication based on copying and buffering of data, resulting in adverse performance costs. An alternative is to interpret the layered model as non-monolithic and *vertically* composed. An object-oriented approach based on class inheritance between well-defined types naturally represents the vertical structure. Applications in which each object is allocated its own thread of control can benefit from this alternative structure since each object can have its own instance of a protocol machine hierarchy. In addition, the protocol machine implementations can be greatly simplified since the vertical partitioning streamlines the bi-directional control flow between service layers and offers opportunities to optimize the layer interfaces based on the code dependency relationship between objects representing each service. Furthermore, it obviates the need for intra-layer multiplexing within the protocol machines, and inter-layer communication is reduced to superclass/subclass method invocations within a single composite object.

In this paper, we focus on our work with an object-oriented implementation of the OSI upper layers, called OOSI (*ooo-zi*), which is in the final stages of development. OOSI is implemented in C++ and is the result of an almost complete re-engineering of the core elements of the ISO Development Environment (ISODE) [Rose91], a widely used research implementation of the upper layer OSI protocols. We think it relevant to explore how object-oriented language features can be used in the implementation of a layered protocol architecture. Constructing layered systems using object-oriented techniques is an area of interest to several others as well. Relatively recent work includes Choices [Campbell 1987], the Conduit [Zweig 1990], and OTSO [Koivisto 1991]. Two reasons motivated us to implement OOSI.

First, we wanted a rich and flexible communications infrastructure on which to build distributed actor-based computations for a new generation of applications. In general, development of OSI-based applications, if undertaken at all, is a difficult and time consuming process. Major difficulties are understanding the use of the multi-option service primitives offered by each layer, use of standard application layer service elements such as remote operations, being facile with the Abstract Syntax Notation One (ASN.1) data representation language, and mapping application services onto an appropriate set of communication services.

Second, having worked with and examined in detail the ISODE code, it became obvious that inheritance, subtyping, and polymorphism could be used to explicate the implicit type structure of the layered protocol machines and improve their efficiency.[1] An aesthetic benefit, which is appealing from a software engineering perspective, is that the type structure reflects the service encapsulation and horizontal composition of the layered model, making it easier to map the components of the model to corresponding elements in the implementation. At the same time, the type structure enforces the proper run-time access control, enables compact vertical composition of methods (inlining), and produces an encapsulated and localized state structure for each instance of the protocol machine hierarchy. The performance improvement results primarily from a simplified control structure induced by the explicit type structure and selective function inlining.

Our current research interest is in rapidly developing new OSI-based applications using an extensible concurrent object-oriented language based on the MIT Actor model [Agha 1986], augmented with reflection and inheritance. The language, called *Rosette* [Tomlinson 1991, Tomlinson 1989], is based on a virtual machine architecture implemented in C++, which has integrated interprocess communication support using both the Internet protocols and the OSI protocols. The OSI interface was originally based on the ISODE, but is being changed to use OOSI. A modular extension of the Rosette language environment, called the *Extensible Services Switch* (ESS), allows rapid implementation and flexible experimentation with new application protocols.

The Carnot Project at MCC used the ESS to implement the Remote Database Access (RDA) protocol and the service agents which provide an RDA interface to a commercialized version of the MCC ORION object-oriented database and other common relational database systems. Other types of service agents have been constructed; for example, a distributed shell. We believe that extensible concurrent object-oriented programming environments like the one offered by Rosette, in conjunction with a rich set of communication abstractions, provide the application layer infrastructure for a much broader open systems technology base than is common in current practice.

We should stress that we are *not* particularly interested in the typical client-server network applications, such as file transfer, electronic mail, and remote virtual terminal. We *are* interested in more powerful application models based on peer-to-peer communication, such as distributed workflow coordination and distributed communicating agents. In programming applications in an inherently concurrent environment like the ESS, it becomes apparent that design choices embodied in the communication infrastructure impact, in a limiting way, the applications infrastructure.

In the remainder of the paper, we discuss our experience with OOSI as an object-oriented rationalization of the upper layer OSI protocols. Section 2 provides a brief overview of relevant object-oriented concepts and briefly introduces the essentials of OSI. Section 3 presents the essential elements of OOSI by discussing how object-oriented features are applied in specific parts of the

1. In addition, inherent weaknesses in the ISODE with respect to truly asynchronous concurrent operation necessitated changes.

implementation. Section 4 offers insights in terms of experience gained and lessons learned. Section 5 summarizes the results of our experience and offers our thoughts on future directions.

2. Models and Terminology

Subsequent sections assume familiarity with concepts from both OOP and OSI. In this section, we briefly review the essential elements from both domains.

2.1 Essential OOP

We assume the reader is familiar with current object-oriented practice. In particular, the reader should be familiar with the notions of inheritance, subtyping, and polymorphism. One should also have an appreciation for the fact that the inheritance mechanism in many languages, including C++, is semantically confused [Cook 1990]. Language inheritance mechanisms are commonly used to construct both subtype relations and code dependency relations. Subtype relations are realized through *specialization inheritance*. Code dependency relations, or code reuse, is realized through *implementation inheritance*. In C++, the same inheritance mechanism is used for constructing both types of relations — it is the programmer's responsibility to separate the two notions in a sensible manner.

The reader should also have an appreciation for the arguments made by Snyder concerning degrees of encapsulation with respect to inheritance [Snyder 1986]. Specifically, the C++ inheritance mechanism permits controlled compile-time relaxation of strict encapsulation and method visibility. It can be argued that controlled relaxation of encapsulation is useful when efficiency requirements are paramount, although doing so imposes limits on the potential for reusability [Scherlis 1986].

The C++ programmer exercises relaxation of encapsulation by inheriting *private*, *protected*, or *public*, while also declaring methods and/or instance variables in a class definition consisting of private, protected, and public clauses. A complete description of the encapsulation relaxation rules is beyond the scope of this paper; we refer the reader to [Ellis 1990]. Subsequent mention of this feature will be in a context simple enough to comprehend.

Polymorphic functions are realized in C++ by a function overloading mechanism that allows multiple function declarations with the same name, but distinct type signatures. Function polymorphism of this form is classified as *ad hoc* polymorphism [Cardelli 1985]. C++ also supports a constrained form of *parametric* polymorphism for functions declared within a class. The *virtual* attribute applied to a function declaration signals to the compiler that a subclass may redefine an inherited function, subject to the inheriting rules just described and signature compatibility rules. The method dispatching mechanism[2] generated by the compiler for an instance of a class contain-

2. Called the *virtual function table* or vtable.

ing virtual function declarations is parameterized based on the type of the object for which a method is dispatched. In addition, C++ supports the notion of *pure virtual* functions. A class containing pure virtual function declarations is considered *abstract* in the sense that there is no implementation associated with a pure virtual function. Some subclass of an abstract class must provide an implementation. The primary purpose of abstract classes and pure virtual functions is to provide the programmer with the ability to define a type but delay the implementation and rely on the parametric method dispatching mechanism to invoke the proper implementation.

2.2 The OSI Model

In case the reader is not familiar with the ISO Reference Model for OSI, this section introduces the basic model and terminology. The next section interjects some pragmatic issues while discussing requirements for implementing portions of the OSI model.

The diagram on the left side of Figure 1 illustrates the basic Reference Model as it is commonly represented. The diagram serves only to illustrate a partitioning of the functional layers into *upper layers* and *lower layers*. The primary purpose of the upper layers is to impose structure onto an otherwise unstructured transport service. The primary function of the lower layers is to provide end-to-end delivery of data, subject to quality of service requirements (e.g., reliable connection-oriented or unreliable connection-less delivery). The distinction is relevant from the perspective that the lower layers are commonly implemented in the operating system and hardware while the upper layers are not.

The OSI model is fundamentally a *layered peer-to-peer* model of interaction. The familiar layered diagram is misleading since it only depicts a layered monolithic structure and omits the peer-to-peer aspects of the model. The diagram on the right side of Figure 1 is a more accurate illustration of the peer-to-peer structure. Interaction among layer entities occurs in two forms: direct and indirect. Direct interaction occurs between vertically adjacent layer entities, related only by the existence of a functional dependency for *service*. Indirect interaction occurs between horizontal peer entities, residing at the same functional layer, which are behaviorally related by a common set of rules or *protocol*. With one exception, indirect interaction can occur only as a consequence of direct interaction.[3]

Direct interaction occurs between service *users* and service *providers*. The Service Access Point (SAP) is the abstraction used by a user to issue a service *request* to a provider. A provider satisfies a service request by indirectly interacting with its remote peer entity using the subset of its behavior specification that implements the requested service. A provider communicates the desired behavior to its peer using an *a priori* protocol definition. This indirect communication is effected by formulating another service request, and so on down to the physical layer. In general, a service request translates into a ordered sequence of function invocations. The (N)-SAP serves as the

3. The one exception being the electrical interaction at the physical layer.

interface point for invoking the Nth function in the sequence. Each (N)-layer function *synthesizes* a Protocol Data Unit (PDU) containing the Protocol Control Information (PCI) representing the behavior change required at the peer entity. A Service Data Unit (SDU) containing the (N)-PDU and the (N+1) SDU formulated by the (N+1)-entity is passed to the (N-1)-layer.

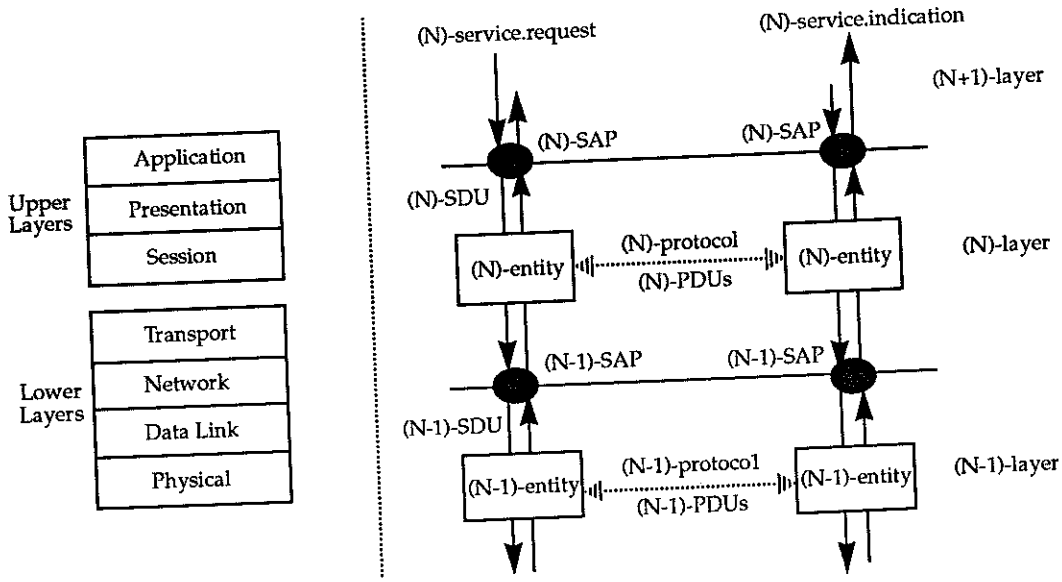

FIGURE 1. Layered Peer-to-Peer Interaction Model[4]

The dual of a service request is an ordered sequence of service *indications* terminating at the peer of the (N)-entity which initiated the request. Each indication in the sequence initiates behavior modification of the entity at that layer as prescribed by the PCI provided as part of the indication. Indications are inherently asynchronous. Each layer entity must be prepared to handle the occurrence of asynchronous indications. Each (N)-layer handling function *analyzes* the (N)-SDU provided as part of the indication, extracts the PCI from the (N)-PDU, and passes the remainder of the SDU to the (N+1)-layer in the form of another indication.

The OSI standards define the service model and specify the details of the protocol for each layer, but leave most implementation issues undefined. Formal implementation agreements, usually at the national level, augment the basic standards by providing implementation guidelines for those areas in the standards which are vague or left to interpretation. Svobodova provides a good presentation on the core implementation issues facing the implementors of OSI systems [Svobodova 1989]. Rose gives a detailed treatment of the pragmatics of OSI using the ISODE as a reference implementation [Rose 1989]. In the next section, we present the requirements for implementing the upper layer protocols in support of object-oriented applications.

4. Adapted in part from [Svobodova 1989].

2.3 Realizing OSI-Based Object-Oriented Systems

Current network-oriented systems consist of a mixture of network services provided by various protocol module combinations. OSI-based applications and services must co-exist with traditional services. An object-oriented application infrastructure must encapsulate both the upper layer OSI services and existing services, typically Internet services. The Internet services are often encapsulated by a RPC mechanism. The usefulness of the RPC paradigm in supporting client-server interactions is well known and is not discussed. In this section, we are primarily interested in the implementation requirements for the OSI upper layers.

Figure 2 illustrates a common multi-protocol architecture used to build distributed object-oriented applications in Unix workstation environments.

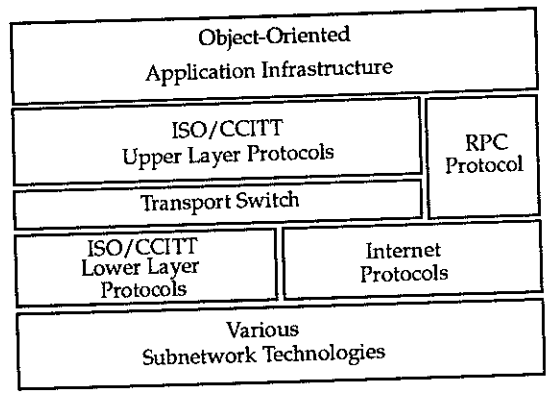

FIGURE 2. Unix-Based Multi-Protocol Architecture

The *transport switch* layer depicted in Figure 2 is not part of the standard OSI model. The concept of transport switching in OSI was first introduced by [Rose 1986]. The switch addresses, in an elegant manner, the pragmatic issue of simultaneously using many underlying transport services arising from different transport/network protocol combinations. The primary function of the switch is to map transport service requests made by a transport user, such as the OSI session service, onto some instance of an OSI transport service. The mechanism implementing a particular transport service instance is most often a kernel-based protocol module accessible via a standard system programming interface.[5] In the common case, the switch can map onto one of a number of OSI transport/network protocol combinations, as well as the non-OSI transport service offered by the well known Transmission Control Protocol (TCP) in conjunction with the Internet Protocol (IP). The mapping onto TCP/IP is done using a convergence protocol defined in [RFC 1006]. In mapping onto TCP/IP, the transport switch implements a lightweight instance of the OSI transport protocol[6] which expects a reliable connection-oriented network service. Thus, using the con-

5. On Berkeley Unix systems the socket interface abstraction is used, while on System V it is the Transport Layer Interface (TLI), perhaps hidden by a socket abstraction.

6. ISO Transport Class 0, or TP0.

vergence protocol defined in RFC 1006, TCP/IP appears to the switch as a reliable connection-oriented network service instead of a stream-oriented transport service.

A significant amount of implementation, experimentation, and performance tuning has been done at the transport layer and below with regard to the Internet protocols [Clark 1989]. The experiences gained with lower layer implementations in the Internet translate well to both the OSI lower layers and upper layers. In particular, protocol engineers know that excessive copying of data severely affects performance, as does layer multiplexing [Tennenhouse 1989]. In general, there has been little attention paid to the fine-tuning of upper layer protocol implementations with regard to the impact on concurrent applications which utilize the upper layer services. In part this is due to the fact that development of new types of OSI-based applications is lagging and in part because a lot of energy has been spent on just making OSI work, in some cases minimally, with traditional applications. It has been shown that architectural choices, and their realization in upper layer implementations, can have a dramatic affect on the ability of the services to meet the type of demands likely to be made by more sophisticated applications [Clark 1990].

Research on lower layer protocol implementation has led to programming techniques for streamlining control flow within a process. From our perspective, the most important contribution to network programming methodology to date is the idea of structuring protocol implementations in terms of *upcalls* [Clark 1985]. The upcall methodology requires that the implementation language support higher-order functions; i.e., provide a mechanism for passing functions, or pointers to functions, as arguments to other functions. Atkins reports on experience using upcalls in a concurrent non-object-oriented language [Atkins 1988]. It is our position, presented in the following section, that inheritance and polymorphism are ideal for implementing the upcall methodology.

Figure 3 illustrates our view of the upper layers in terms of a *vertical* partitioning of protocol machines (PMs) and service access points (SAPs), application entities (objects), and the kernel entities representing a particular instantiation of the lower layers required to meet the quality of service demands of an application entity. Vertical partitioning permits several instances of the protocol machine hierarchy to exist within a single process. The diagram depicts a single application process with multiple application entities, each with its own ordered set of service access points and protocol machines down to a *Virtual Transport Service Access Point* (VTSAP) that encapsulates the transport switch. The switch interfaces to a real transport service residing in the kernel entity. The vertical structure facilitates multi-threaded application entities since each slice through the layers does not interfere with any other slice. Each protocol machine can be implemented such that it need not concern itself with multiplexing since it has a single downcall service access point and a single upcall service access point. However, each slice must guarantee internal consistency with respect to downcalls and upcalls since they may occur asynchronously. The guarantee is typically achieved by enforcing mutual exclusion on the state of each protocol machine during either an upcall or a downcall.

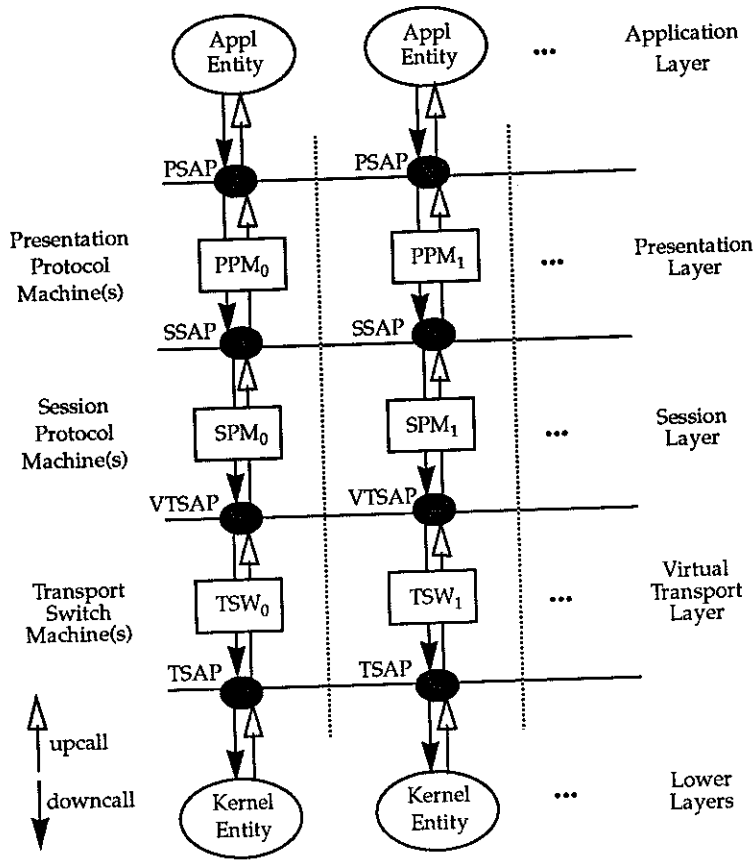

FIGURE 3. Vertical Partitioning of the Upper Layers

The primary goal of the upper layers is to deliver structured application entity data to/from the kernel entity representing the lower layers as efficiently as possible, while maintaining the consistency of the protocol machine instances at each layer. The control flow through the layers is asynchronous and bi-directional. Control flow is initiated by either an application layer entity generating data, or a kernel entity receiving data from the network. As already discussed, the usual action of a protocol machine is to formulate a header containing a relatively small amount of protocol control information, prepend the header to any data passed as part of the service invocation, update some local state, and then pass the data onto the next lower layer. Similarly, data received from the network causes the kernel entity to initiate the upward control flow by issuing an indication event to the application process in the form of an interrupt. An application I/O interrupt dispatching mechanism handles the indication and triggers an application process upcall sequence starting at an instance of a transport switch and ending at the corresponding application entity.

In the following discussion on OOSI, we explain how object-oriented language features are used to facilitate and enhance the implementation of the protocol machine and service access point abstractions presented in the Figure 3.

3. Elements of the OOSI Design

Elements of the OOSI design are predicated on the notion of a *well-behaved* or *trusted* user of a service layer. This notion does not imply an ad hoc relaxation of layer encapsulation. To the contrary, it exploits the fact that the user of a layer is most often another encapsulated object that is well-behaved and also expects its clients to be well-behaved, or rigorously defends itself against misuse. We might argue that the object-oriented paradigm induces recursively well-behaved objects up to some access point.

3.1 Class and Inheritance Structure

The OOSI implementation structure closely models the vertical partitioning of the upper layer architecture, depicted previously in Figure 3. Layering induces a downward functional dependency relationship between a service user and a service provider. The functional dependency relationship induces an inheritance hierarchy, depicted in Figure 4, that is inverted with respect to the layer hierarchy.

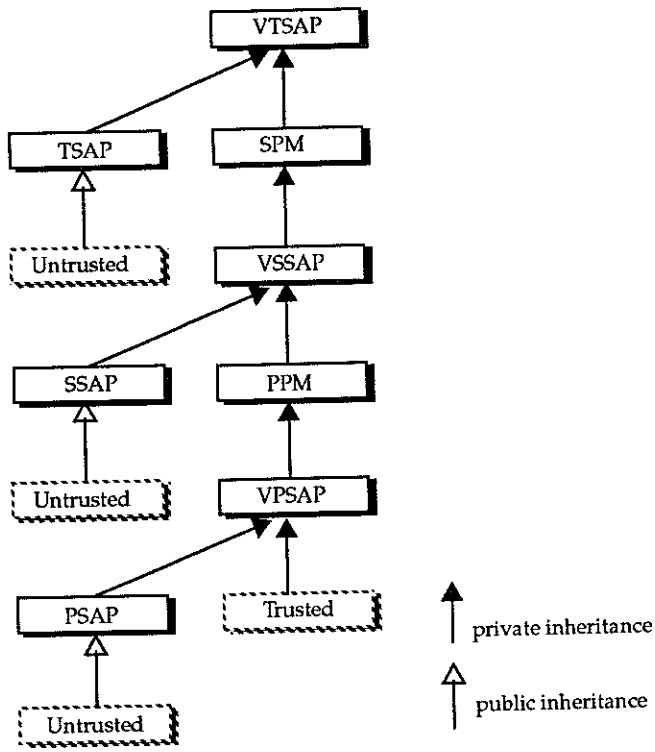

FIGURE 4. Primary OOSI Inheritance Hierarchy

Each layer is represented by three classes:

- a **Protocol Machine** (PM) class.
- a **Virtual Service Access Point** (VSAP) class inheriting privately from the PM class.
- a **Service Access Point** (SAP) class inheriting privately from the VSAP class.

As depicted in Figure 4, the top-most class in the OOSI hierarchy represents the VTSAP interface to the transport switch. For reasons that will be explained later, the switch itself does not occur in this structure. The hierarchy continues downward with PM, VSAP, and SAP classes representing each layer. Each PM inherits from a VSAP instead of a SAP since each PM is trusted to be well-behaved. The VSAPs are used to compose trusted entities. The introduction of a VSAP derives from the observation that the typical user of a layer is a well-behaved entity, usually another protocol machine. The VSAP implicitly makes the assumption that the well-behaved entity is either used by yet another well-behaved entity, or rigorously enforces good behavior. The assumption made by the VSAP translates in the implementation to the relaxation of the rigor with which the interface to the layer is checked. Interface checking typically consists of guaranteeing that the PM at that layer is in a state allowing the requested operation and guaranteeing mutual exclusion during the execution of the operation which is part of a downcall or upcall function sequence. Rigorous enforcement is the function of the SAP class. The choice of which interface to use when accessing a service layer is a choice made by the programmer. In the best scenario, a service user is a provably well-behaved object; however, in our experience, we find that lack of such a strong guarantee is not detrimental.

For example, the Presentation PM (PPM) inherits from the Virtual Session SAP (VSSAP) instead of the SSAP since the PPM is a trusted Session user. If a user of the Presentation service is trusted, then it inherits from the VPSAP, otherwise the user is untrusted and inherits from the PSAP. The inheritance hierarchy depicted in Figure 4 can be extended into the application layer by defining application layer protocol machines which inherit, at the discretion of the implementor, from either the PSAP or the VPSAP. In either case, the private inheritance mechanism of C++ provides the necessary access control to ensures that total encapsulation of a layer is maintained.

The ability to forgo rigorous checking for each operation at each layer achieves better performance as the result of a simplified control flow for downcalls. Furthermore, the VSAP is made "virtual" by inlining its functions, thereby avoiding procedure call overhead. At run-time we achieve a collapsing of the layers resulting in concatenation of each protocol machine's state, and all machines share the same method dispatching structure (the vtable). Note that layer encapsulation is still guaranteed because of the protection induced by the private inheritance relationship between a VSAP type and its corresponding PM type. A VSAP separates the layer interface from the protocol machine implementation. The merits of function inlining are often debated. In the case of VSAP functions, the modest increase in code size seems insignificant in comparison with the gain in type abstraction, protection, and separation of concerns without the cost of an extra function call.[7]

One might first think that imposing an inheritance hierarchy on the layer structure is artificial — that all that is required is strict encapsulation. However, there is more going on than just using a class to encapsulate a layer. Using inheritance in this context results in two distinct advantages.

7. One can surmise that to accomplish a similar feat in traditional C would require at least two functions: a function declared static and a function declared external with both function definitions occurring within the same file scope.

First, the type structure induced by the inheritance hierarchy directly models the layer structure. The correspondence between the code and the model enhances the understandability of the code. Networking code is often notoriously difficult to maintain and it is important to be able to examine the code and have it reflect the mental model one has concerning its structure. Second, recall that control flow through the layers is bi-directional and asynchronous. Downcalls/upcalls through the layers translate into superclass/subclass method invocations within a single composite run-time object. Downcalls become direct superclass method invocations, via the method dispatch table, from one PM to another, stream-lined by the fact that a PM inherits from a VSAP and all VSAP methods are inlined. The technique for implementing upcalls relies on parametric polymorphic functions.

A PM has to be prepared to handle an asynchronous upcall from the adjacent lower layer. After analyzing the PCI and updating some local state, the PM issues an upcall to the adjacent higher layer. The traditional technique for implementing upcalls is that a layer provides the adjacent lower layer with a set of function pointers, via a special binding operation, which the lower dereferences and uses to invoke a higher layer function as part of an upcall function sequence. This technique works; however, the disadvantages are that it is difficult to follow the control flow when examining the code and the upcall functions are either untyped or weakly typed. To invoke an upcall in OOSI, a PM method calls a typed pure virtual function, whose type signature occurs in a C++ *protected* clause in the PM and is repeated in the VSAP and SAP interfaces. The method dispatch mechanism generated by the compiler does the dereference based on its type rules and invokes a method implemented in the subclass PM. Depending on the compiler, performance may be slightly worse if two pointer dereferences are required to make the call.[8]

The advantage of the polymorphic function approach used in OOSI is that all upcall functions are strongly typed and the code is more comprehensible. The disadvantage of the OOSI approach is that it is inherently more static since the method dispatch table is determined at compile time. The traditional approach is more dynamic since the binding can be changed at run time. However, it is not generally the case that the binding between protocol machines changes at run time. We have experimented in OOSI with providing an abstract **Indication** class at each layer which declares the set of upcall type signatures required for a layer. The Indication class can be specialized to provide various implementations. Instances of the Indication class could then be dynamically bound to a protocol machine. The final version of OOSI may support both mechanisms.

3.2 The Transport Switch

The transport switch must present a Session user with a single consistent transport abstraction irrespective of the *type* of the transport mechanism and the *implementation* of the mechanism. The transport switch presents an interesting opportunity to utilize subtyping in useful manner. For discussion purposes, we describe only the connection-oriented portion of the switch.

8. In C++, the *this* pointer, which is either on the stack or in a register, and the vtable pointer.

The transport switch is represented by the subtype hierarchy depicted in Figure 5.

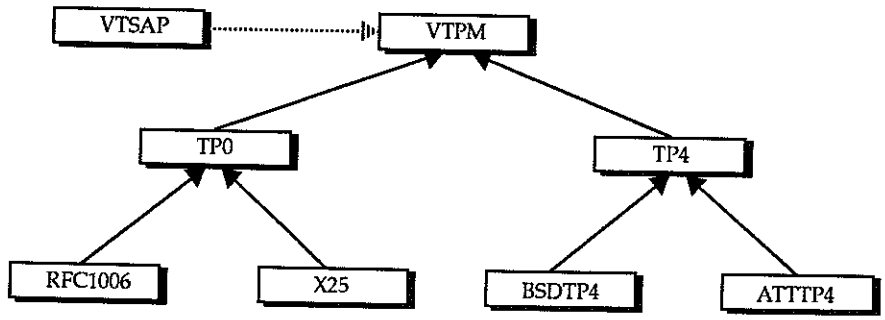

FIGURE 5. Transport Switch Hierarchy

The top-most class in the hierarchy is an abstract class called the *Virtual Transport Protocol Machine* (VTPM). The VTPM defines a set of pure virtual functions that each subclass must implement. There are two subclasses representing the two principal types of the OSI connection-oriented transport service: Transport Class 0 (TP0) and Transport Class 4 (TP4). The TP4 is itself abstract and is further specialized into either a BSD (BSDTP4) or an AT&T (ATTTP4) implementation. The BSDTP4 class maps service requests onto the Berkeley Unix TP4 kernel module via sockets while the ATTTP4 class maps onto the AT&T TP4 kernel module via the TLI. In general, a given system only provides one type of TP4 service.

The TP0 is a concrete class that implements the ISO TP0 protocol, but is unique in that it is only a partial implementation. The TP0 class implements the basic TP0 protocol but defines an additional set of pure virtual functions which the RFC1006 and X25 subclasses must implement in order for the TP0 protocol machine to be complete. The TCP subclass implements the RFC 1006 convergence protocol mapping TP0 onto the TCP, treating TCP/IP as a connection-oriented network service. This class also encapsulates the actual kernel interface. The X25 class implements a mapping of TP0 onto a CCITT X.25 connection-oriented network service. Although instances of the VTPM hierarchy are treated uniformly, they offer distinct services. For example, TP4 offers an expedited data transfer service, while TP0 does not. However, an RFC1006 object augments the TP0 service by providing expedited data transfer using the out-of-band data channel provided by the TCP.

The VTSAP relates an instance of the primary OOSI inheritance hierarchy presented in Figure 4 with an instance of the switch. Two separate hierarchies are required because the determination of which type of switch to instantiate is made at connection establishment time by the VTSAP based on the quality of service requirement of the initiating application layer entity and a reachability requirement based on the network address of the peer entity. Once the determination is made, say for an RFC1006 instance, an object is instantiated by and bound to the VTSAP. Since the VTSAP is inlined into the SPM, transport service requests are made directly by the SPM to a switch instance via a VTPM-typed pointer.

4. Experience Gained, Lessons Learned

In the previous section we presented the essential structure of the OOSI implementation to demonstrate the usefulness of object-oriented techniques in the implementation of the OSI layered protocol architecture. The process of arriving at that structure seems in retrospect as important as the structure itself. In this section we summarize our experience designing and implementing OOSI as a re-engineering of the C implementation of the ISODE. We present our views of the usefulness of structuring layered protocol architectures with respect to inheritance, subtyping, and polymorphism. We also report on our experience reusing existing C code.

4.1 Inheritance, Subtyping, and Polymorphism

The primary OOSI inheritance hierarchy is interesting from the perspective that it satisfies both the functional dependency relationship required to support downcalls, while at the same time satisfying the specialization relationship required to implement upcalls. That is, a subclass (higher layer) depends on its superclass (lower layer) for code reuse while the superclass depends on the subclass to implement typed indication handlers by specializing superclass methods. One might argue that we are misusing inheritance, but the structure we present seems elegant and matches intuition about how control flows up and down through the layers. Parametric polymorphism seems particularly powerful in allowing us to structure the upcalls as superclass to subclass method invocations.

An area we are currently researching relates to synchronization control in concurrent object-oriented languages with inheritance. The basic idea is that the set of methods visible in the interface to an object can be computed and controlled based on the internal state of an object. Various techniques have been proposed in different language contexts. Tomlinson and Singh implemented *Enabled Sets* in Rosette [Tomlinson 1989], while Kafura and Lee implemented *Behavior Abstraction* in ACT++ [Kafura 1989]. Protocol machines have similar requirements for synchronization control. By incorporating both the downcall and upcall methods in the class interface, rather than separating them, we are in a position to exploit future developments with regard to controlling method access via an object interface. We are actively seeking a suitably efficient mechanism for C++. We have experimented with various techniques external to the compiler, all of which are too heavyweight for use in protocol machines.

In the early stages of the OOSI design, we experimented with using multiple inheritance in structuring the transport switch. We discovered, after a bit of rational thought, that single inheritance was sufficient and easier to understand. For example, it seemed useful to structure the transport switch by also specializing classes based on *passive* and *active* types. In the protocol arena, a passive protocol machine does not participate in data transfer, it simply listens for connection requests and creates an instance of an active protocol machine to handle the connection. The result turned out to be unwieldy for the small gain of having the somewhat artificial active-passive type distinction. The switch subtype hierarchy presented above is simple and easy to understand. Behavior abstractions, once implemented efficiently, should enable the active-passive distinction.

Although not presented, we used inheritance to structure the different types of Protocol Data Units used at each layer into subtype hierarchies. We observed in the ISODE implementation that PDUs are typically represented by discriminated unions. Type discrimination via case analysis is often repeated in different parts of a protocol machine when processing incoming PDUs. By introducing PDU type hierarchies, we perform PDU analysis once when an indication is received and instantiate an instance representing the PDU. Subsequent operations on a PDU are invoked via a supertype pointer that relies on the method dispatch table generated by the compiler to effect type discrimination. The result is simplified code. It is unclear at this point whether or not the control flow is simplified to the degree that a significant performance advantage is achieved over the case analysis code that would otherwise have been written.

4.2 Reuse

The result of imposing an inheritance and type structure on the existing ISODE code was an almost total rewrite of the core elements. Imposing the object-oriented type structure necessitated major changes to much of the existing C code because the control flow and method of interaction changed significantly. It was not sufficient to simply wrap the C code in a C++ abstraction since we were after efficiency as well as improved structure. In our experience, true object-oriented re-engineering means that traditional C code cannot be effectively reused.

A particularly annoying aspect of the ISODE, and other libraries we use, is that they implicitly assume that a process is single threaded. The tendency is to implement service libraries which define a policy of handling interrupts or asynchronous activity in a polled fashion as opposed to a completely asynchronous interrupt driven manner, which inhibits the incorporation into other software components which wish to make other policy decisions. The ISODE is primarily based on a polling philosophy. To some degree, this is a result of the programming paradigm offered by C in which there is no explicit concept of concurrency.

For example, the Rosette language environment is inherently concurrent. The Rosette virtual machine implements very lightweight actor threads called strands. Any blocking behavior, say as the result of a foreign function call to a C library routine which blocks, leaves the virtual machine blocked. We are forced to perform surgery on such libraries to rewrite the asynchronous handling code. We suggest that implementors begin to view their systems as a component of a multi-threaded process and not assume that they can block the process in an ad hoc manner.

5. Summary

We have presented elements of the design of OOSI, an object-oriented rationalization of the upper layer OSI protocols. OOSI uses inheritance, subtyping, and polymorphism to impose a type structure that captures in code the layered protocol architecture without compromising the performance of the protocol machines at each layer. The type structure enhances the understandability of the protocol machine implementations and elegantly captures the asynchronous bi-directional

control flow inherent in a peer-to-peer model such as the OSI model. The primary OOSI inheritance hierarchy relies on the concept of a Virtual Service Access Point or VSAP to facilitate composition of protocol machines without compromising encapsulation. Function inlining is used to make the run-time existence of a VSAP virtual.

The VSAP concept is similar to, but distinct from, other notions of "virtual protocols". Using inheritance and polymorphism to compose protocol machines such that upcalls and downcalls through the protocol layers translate into subclass and superclass method invocations seems particularly elegant and useful, although it is inherently static in nature. Others have experimented with flexible protocol composition mechanisms at the lower layers [Hutchinson 1989], [Tschudin 1991]. The concept of a more flexible protocol stack is achievable in OOSI using object-oriented techniques and may be incorporated in a future version.

Acknowledgments

Phil Cannata of Bellcore, director of the Carnot Project at MCC, deserves acknowledgment for creating and maintaining the environment in which this work was done. The existence of the ISODE as a research tool, created principally by Marshall Rose, contributed significantly to educating us on the pragmatics of OSI.

References

[Agha 86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*, M.I.T Press, 1986.

[Atkins 1988] M. Stella Atkins. "Experiments in SR with different upcall program structures," *ACM TOCS*, 6(4), November 1988, pp. 365–392.

[Campbell 1987] Roy H. Campbell, Gary M. Johnston and Vincent F. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," *Operating Systems Review*, No. 21, July 1987, pp. 9—17.

[Cardelli 1985] Luca Cardelli and Peter Wegner. "On understanding types, data abstraction, and polymorphism," ACM Computing Surveys, 17(4), December 1985, pp. 471–522.

[CCITT 1989] *Data Communication Networks — Open Systems Interconnection (OSI) Model and Notation, Service Definition*, Recommendations X.200-X.219, Blue Book Series, The International Telegraph and Telephone Consultative Committee, Geneva 1989.

[Clark 1985] David D. Clark. "The structuring of systems using upcalls," *10th ACM Symposium on Operating System Principles*, December 1985, pp. 171–180.

[Clark 1989] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. "An analysis of TCP processing overhead," *IEEE Communications Magazine*, June 1989, pp. 23–29.

[Clark 1990] David D. Clark and David L. Tennenhouse. "Architectural considerations for a new generation of protocols," *ACM SIGCOMM'90*, pp. 200–208.

[Cook 1990]. William R. Cook, Walter L. Hill, and Peter S. Canning. "Inheritance is not Subtyping," *Seventeenth Annual ACM Symposium on Principles of Programming Languages*, January 1990, pp. 125–135.

[Ellis 1990] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Hutchinson 1989] Norman C. Hutchinson, larry L. Peterson, Mark B. Abbott, and Sean O'Malley. "RPC in the x-kernel: evaluating new design techniques," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989, pp. 91–101.

[Kafura 1989] Dennis G. Kafura and Keung Hae Lee. "Inheritance in actor-based concurrent object-oriented languages," *ECOOP'89 Conference Proceedings*, Cambridge University Press, 1989, pp. 131—145.

[Koivisto 1990] Juha Koivisto and Juhani Malka. "OTSO - an object-oriented approach to distributed computation," *Usenix C++ Conference Proceedings*, April 1991, pp. 163–177.

[RFC 1006] Marshall T. Rose and Dwight E. Cass. *RFC 1006 — ISO Transport Services on top of the TCP, Version 3*, Internet Request For Comments, May 1987.

[Rose 1986] Marshall T. Rose. "OSI transport services on top of the TCP," *Computer Networks and ISDN Systems*, 12(3), 1986.

[Rose 1989] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI*, Prentice-Hall, 1989.

[Scherlis 1986] William L. Scherlis. "Abstract data types, specialization, and program reuse," in *International Workshop on Advanced Programming Environments*, Springer Verlag, 1986

[Snyder 1986] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages," *ACM OOPSLA'86 Conference Proceedings*, September 1986, pp. 38–45.

[Svobodova 1989] Liba Svobodova. "Implementing OSI systems," *IEEE Journal on Selected Areas in Communications*, 7(7), September 1989, pp. 1115–1130.

[Tennenhouse 1989] David L. Tennenhouse. "Layered multiplexing considered harmful," *Protocols for High-Speed Networks*, North-Holland, 1989.

[Tomlinson 1989] Chris Tomlinson and Vineet Singh. "Inheritance and synchronization with enabled-sets," *ACM OOPSLA'89 Conference Proceedings*, October 1989, pp. 103–112.

[Tomlinson 1991] Chris Tomlinson, Mark Scheevel, and Vineet Singh. *Report on Rosette 1.1*, MCC Technical Report ACT-OODS-275-91.

[Tschudin 1991] Christian Tschudin. "Flexible Protocol Stacks," *ACM SIGCOMM'91 Conference Proceedings*, September 1991, pp. 197–205.

[Zweig 1990] Jonathan M. Zweig and Ralph E. Johnson. "The conduit: a communication abstraction in C++," *1990 Usenix C++ Conference Proceedings*, April 1990, pp. 191–203.