A base for portable communications software

by S. H. Goldberg
J. A. Mouton, Jr.

The emerging international standards for interconnecting computers will be important in IBM's future plans. The Open Systems Interconnection (OSI) protocols are already part of IBM's Systems Application Architecture® (SAA™), implying that they will be implemented across the dissimilar SAA operating systems. Building these complex OSI protocols is costly, and additional expense is involved in verifying conformance and interoperation with other systems. "Porting" a common implementation of these protocols to all SAA systems offers major cost savings, but the differences between systems and the need for high-performance, robust implementations poses problems. The OSI/Communications Subsystem Base solves many of these problems in a general way that may apply to other layered protocols and other systems. The Base provides all necessary operating system services to support the layered communications protocol machines of OSI and allows access to the I/O services of the native operating system as required. This paper discusses the sophisticated communications-oriented environment provided by the OSI/Communications Subsystem Base, which includes multiple threads, back-pressure flow control, resource monitoring, layer modularity, and steps to minimize process switches and data copying. The paper is addressed primarily to systems engineers and communications architects interested in OSI and portability in general.

Modern communications protocols have a layered architecture. Systems Network Architecture (SNA), Transport Control Protocol/Internet Protocol (TCP/IP), and Open Systems Interconnection (OSI) protocols are defined as layers of protocol, each addressing a part of the overall communications function. OSI is a family of layered communications protocols defined by international standards.[1] The OSI Reference Model, used as a basis for development of OSI standards, is a descriptive model of an arbitrary communications system, including all required functions. The model defines seven layers:

1. The Physical Layer, for physically connecting communicating stations
2. The Data Link Layer, for structuring data on a physical connection
3. The Network Layer, for routing and relaying traffic worldwide
4. The Transport Layer, for guaranteeing end-to-end data transmission
5. The Session Layer, for connecting applications and structuring the dialog between them
6. The Presentation Layer, for standardized data representation
7. The Application Layer, for communicating with specific applications

OSI is becoming internationally accepted as the protocol of choice to interconnect computer systems in a multivendor environment. It is anticipated that most computer vendors will implement

OSI protocols. IBM is a leader in this area and has included OSI protocols as part of its Systems Application Architecture® (SAA™).[2]

Software portability. Early efforts to implement communications protocols usually produced code specific to each target system. In all cases, protocol implementations must "interoperate" with implementations in other systems in order to be successful. Their main goal, after all, is to allow communication and useful sharing of work with any other system implementing the same protocol. With system-specific implementations, differences in their operation at times led to failure of communication, especially in unusual circumstances. Additional testing and increased development and maintenance costs were the result. This problem of interoperation is even greater with OSI. The basic purpose of OSI is to allow interoperation of different vendors' computer systems. Since the number of OSI implementations is potentially far greater than those for a proprietary protocol, the costs of development and maintenance could become prohibitive. IBM's solution is to implement each required OSI protocol once and "port" (move in whole) these implementations to all SAA systems.

A basic concept used in modern language compilers is that of a run-time environment.

The advantages of "porting" programs has been recognized for many years. The development and standardization of high-level languages which were supported across different systems gave application developers the ability to write programs in such a way that they could be executed on systems with different internal architectures without major rework. A basic concept used in modern language compilers is that of a *run-time environment*. The typical program written in COBOL or C (for example) will consist of two components when it runs:

- Routines generated directly from the source file
- Routines of a general form that are supplied with the compiler to perform broad, complex, or system-specific operations

These latter routines make up the run-time environment associated with the language itself. Programs written in a given language share the same run-time routines, using only those that are required. The run-time environment presents a standardized interface in any system—the same language instructions are used to invoke them. However, these routines have to be implemented separately on each machine in a system-dependent way.

Run-time routines supplied with standard compilers are often very general and designed to support relatively simple applications that require few system services. Applications needing special functions of an operating system, such as control of tasking operations, required system-specific routines to be developed and used to augment the provided run-time environment.

Another important step in the history of portability is the development of the UNIX® operating system. The UNIX system was designed to port easily to different systems. Its portability is based on the C language, and the *kernel* may in some ways be considered an extension to the concept of a run-time environment. The kernel is a small component that includes all of the resource management routines of the operating system. The kernel provides a defined interface to the UNIX *shell* and application programs, the same interface in any system to which the UNIX system is ported. The kernel is implemented in a system-specific way, and developing a kernel implementation is a key part of porting the UNIX system to another system. The rest of the UNIX system is developed in C, and so is portable to a new system once the kernel interfaces have been provided.

Porting the protocol implementation is a key concept in the design of the OSI/Communications Subsystem. The OSI/Communications Subsystem is the IBM program product that implements the middle layers of OSI (the top of Layer 3 through the bottom of Layer 7) on IBM SAA systems—Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA™), Virtual Machine/Enterprise Systems Architecture (VM/ESA™), Operating System/400® (OS/400®), and

Operating System/2® (OS/2®).[3] In the OSI/Communications Subsystem the portability concepts described above are used and enhanced through the use of a system-dependent component called the Base, which is described in this paper. The OSI/Communications Subsystem is available for MVS/ESA, VM/ESA, OS/2, and OS/400.

The IBM Communications Systems Programming Development Laboratory–West Coast in Palo Alto, California, has been developing portable products for over 10 years. Before developing the OSI/Communications Subsystem, it delivered COBOL compilers, sort and merge programs, and communications systems based on the concepts of system-dependent and system-independent components.

Layered implementations. Layered communications protocols have suggested layered implementations from the beginning. The layering of protocol specification contributes to the simplicity of each layer by separating functions cleanly. The OSI standards use this technique. A consideration in developing OSI implementations is whether to reflect layering in the implementation.

Implementing with cleanly separated protocol layers has several advantages:

- A layer component is smaller and simpler than a component that includes several layers, and simpler components generally are easier to build and less error-prone.
- Each protocol layer can be designed separately without considering characteristics of the other layers.
- OSI lends itself to providing interfaces at several layers, and Transport, Session, Presentation, and Application layer interfaces are in use today; breaking into the middle of an unlayered implementation to provide these interfaces can be difficult.
- Some protocol conformance tests have been formulated for individual layers, and performing these tests requires access to a layer without use of layers above it.
- As changes in OSI standards are made separately on each layer, reflecting these changes as they occur is easier when layers are separate.
- Separation of layers (including state information) can allow implementations to better isolate the effects of failures.
- A layered structure with isolated layer components lends itself well to the concepts of software engineering.

However, layered implementations are perceived to provide worse performance characteristics than implementations whose internal structure does not reflect protocol layering. This perception is a result of the cost of the hard interfaces between layers. The costs associated with a hard interface might include the following:

Performance, reliability, and serviceability are all desirable features of good protocol implementations.

- Interfaces in which common data are not shared among the modules require parameterization of all passed data. Building and passing parameter lists is more costly than referring to common data areas.
- Where layers cannot address the same storage areas, all parameters (including the data buffers) must be moved between the separate address spaces in addition to being passed in parameter lists.
- Where layers are implemented as separate processes, the parameters and data buffers are passed using some type of interprocess signal (such as message queuing) rather than by direct procedure call.
- Operating systems typically have a significant cost associated with the process switching in such a design.

Performance, reliability, and serviceability are all desirable features of good protocol implementations. The performance advantages of unlayered implementations often come at the cost of reliability and serviceability. So rather than choosing one approach against the other, a balance must be struck between them. Modern multiprocessor systems such as the System/390™ have thrown another complication into this balance. In a mul-

tiprocessor system, a single process may be limited in the amount of system power that can be used; several processes are needed to take full advantage of these systems. A tradeoff must be made against the path length cost of supporting several processes and the real performance gain achieved by concurrent processing.

Several attempts at this balance have been made, a notable one being D. Clark's Swift operating system.[4] Clark attempts to reduce the cost of interlayer interfaces by defining them as procedure calls, mostly "upcalls" from lower layers to their clients. Layers are defined as groups of related subroutines called "multitask modules," with specific procedures for interaction with other layers. Processes are freed from layer boundaries and are used to carry work up and down the layers in a synchronous way. Mapping of processes to operations is performed by either the lower or upper layer routines or both, and can be changed at a late design stage. Common addressable storage is used to hold shared state information, and monitor locks are used to serialize access to these data.

Clark's approach reduces the cost of layer boundaries to a minimum, using pure procedure calls. However, the sharing of data among all layers loses some of the advantages of layer interfaces, and he discusses the impacts of failures in one layer that can affect the state of other layers.

With the Base, a layered structure has been implemented using the techniques of software engineering, and an attempt has been made to retain most of the good characteristics of hard layer interfaces while reducing the performance cost of these interfaces. The parameterization of a clean interface has been retained, but the Base has been designed to reduce process switching and the overhead of interprocess signaling and to eliminate movement of buffered data between layers. As in the Swift approach, the Base separates tasking considerations from layer implementation and provides common control block and buffer storage to reduce data movement. Compared to Swift, the Base retains a harder layer interface but still provides synchronous flow between the layers whenever possible. Some performance is traded for robustness and portability; however, many of the performance problems Clark identifies are solved.

Several existing protocol implementations use a layered approach. The I/O facilities of the IBM Distributed Processing Programming Executive (DPPX)[5] exemplify an early implementation of SNA that mirrors SNA layering. Another example is the STREAMS facility of UNIX System V Release 3.[6] STREAMS provides an environment for implementing layered communications protocols in a portable way within the UNIX environment.

With the Base, a layered structure has been implemented using the techniques of software engineering.

As with many other communications protocols, the OSI protocols are specified as *finite state machines*. Finite state machines are conceptual entities that support a finite set of possible internal states, accept a predefined set of inputs, and have a well-defined operation, output, and resulting internal state for each possible input received in each possible state.

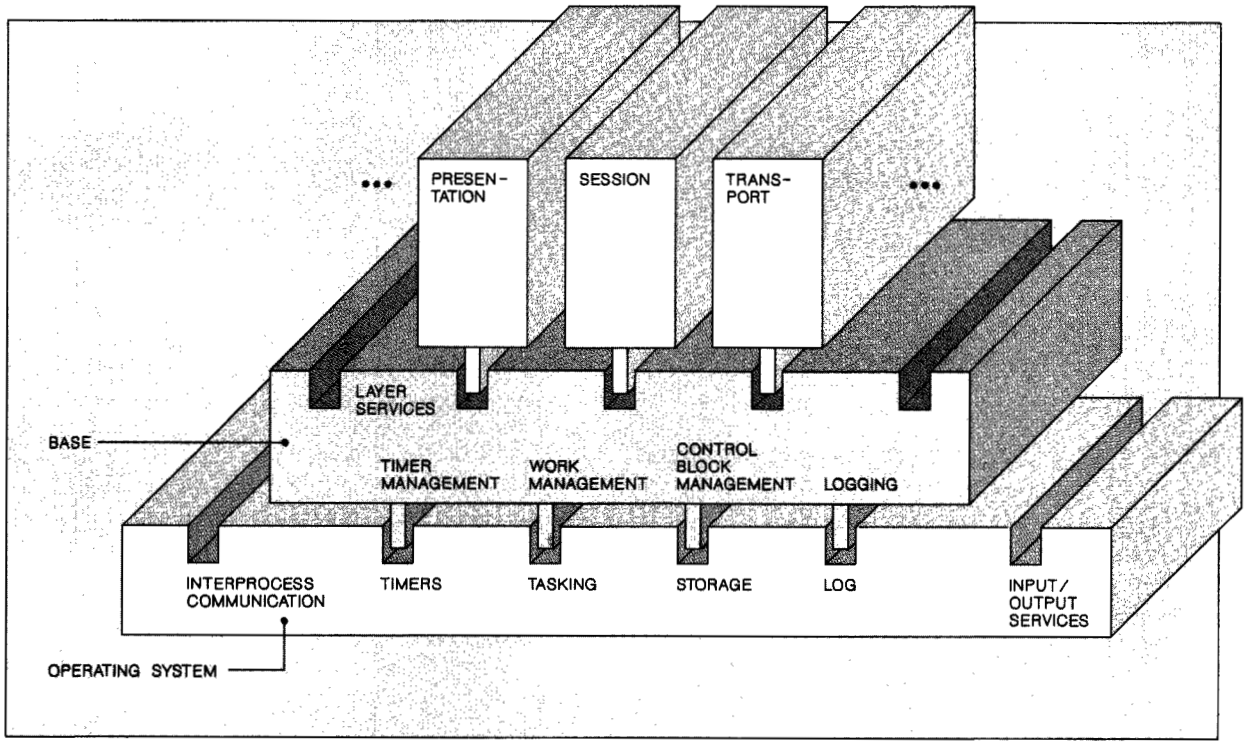
The term *protocol machine* is used in this paper to refer to any implementation of a protocol finite state machine as a component of a communications system.

This paper describes the techniques used by the OSI/Communications Subsystem to provide a portable OSI solution for SAA environments. Attention is focused on the architecture and concepts of the OSI/Communications Subsystem Base (*the Base*), which provides the environment to allow protocol machines to be ported. The paper presents an overview of the Base, a description of its architectural concepts, and a discussion of its service categories. Finally, it outlines experiences in implementing the Base in the OSI/Communications Subsystem products.

The OSI/Communications Subsystem Base

Overview of the Base. The Base may be viewed as a special-purpose run-time environment which is

Figure 1 Plugging layers into the Base



itself portable to different operating systems environments on the same or different hardware. One goal for the Base is to exploit the capabilities of each system, not just to execute successfully in each. The Base concentrates its key interactions with the local operating system into a small amount of code giving Base implementers significant freedom to capitalize on the capabilities of the target systems, especially the tasking structure and storage utilization. Another goal is to accommodate the OSI Reference Model and layer standards, as well as other layered communications protocols. The Base must be able to handle the wide range of options and profiles supported in OSI as well as adapt to the changing environment of developing OSI standards. A third goal is to facilitate the use of common test tools and scenarios across OSI products. A final goal is to meet the performance requirements of communications systems while implementing the other goals.

Architecture. An operating system may be viewed as a service provider, offering services to its users, which are programs. Each operating system is characterized by its own interface to the provided services and in fact may be tailored to a particular class of users or applications. As depicted in Figure 1, the OSI/Communications Subsystem *Base* utilizes whatever services are offered by the underlying operating system to in turn provide uniform services to the OSI/Communications Subsystem *layers*.[7] Different implementations of the Base provide the same layer service interfaces so that the layers become independent of the operating system service interfaces. The layers implement the protocols of OSI—for example Presentation, Session, or Transport—while the Base provides the independent platform necessary to isolate the layers from the idiosyncrasies of each operating system.

The services provided by the Base are tailored to meet the special needs of communications protocols. OSI/Communications Subsystem layers are transaction-oriented protocol machines. They need to be invoked for execution, to maintain control blocks, and to buffer data. They require timer services, message logging, and tracing. Al-

though most layers perform processing functions only, certain layers need access to the operating system I/O services and interprocess communication services to get at communications links, client applications, or system disk storage.

Using an analogy of hardware components, we can compare the Base to a "motherboard" and the layers to "processing cards." The Base serves as the controller of all attached cards and provides them with a rich set of system services. The layers in turn are like processing cards plugged into the motherboard, where each processing card is an implementation of an OSI protocol machine. (To carry the analogy further, we could consider the operating system as the "system enclosure," which provides power to the motherboard and supports interface plugs to the cables in the outside world.)

The Base serves as the controller of all attached cards and provides them with a rich set of system services.

The Base supports as many layers as needed for OSI or another protocol set and allows any layer to obtain the services of any other layer. The Base interfaces are general, not OSI-specific. The OSI implementation is isolated to the layers of the OSI/Communications Subsystem, and SNA or TCP/IP layers could be implemented as well within the Base. In the OSI/Communications Subsystem, the following OSI protocols are implemented as layers:

- Management functions, including Common Management Information Protocol (CMIP) and X.500 Directory Access Protocol (DAP)
- Association Control Service Element (ACSE) and Presentation (combined into one layer)
- Session
- Transport
- Network, including Connectionless-Mode Network Protocol (CLNP) and Connection-Oriented Network Services (CONS)

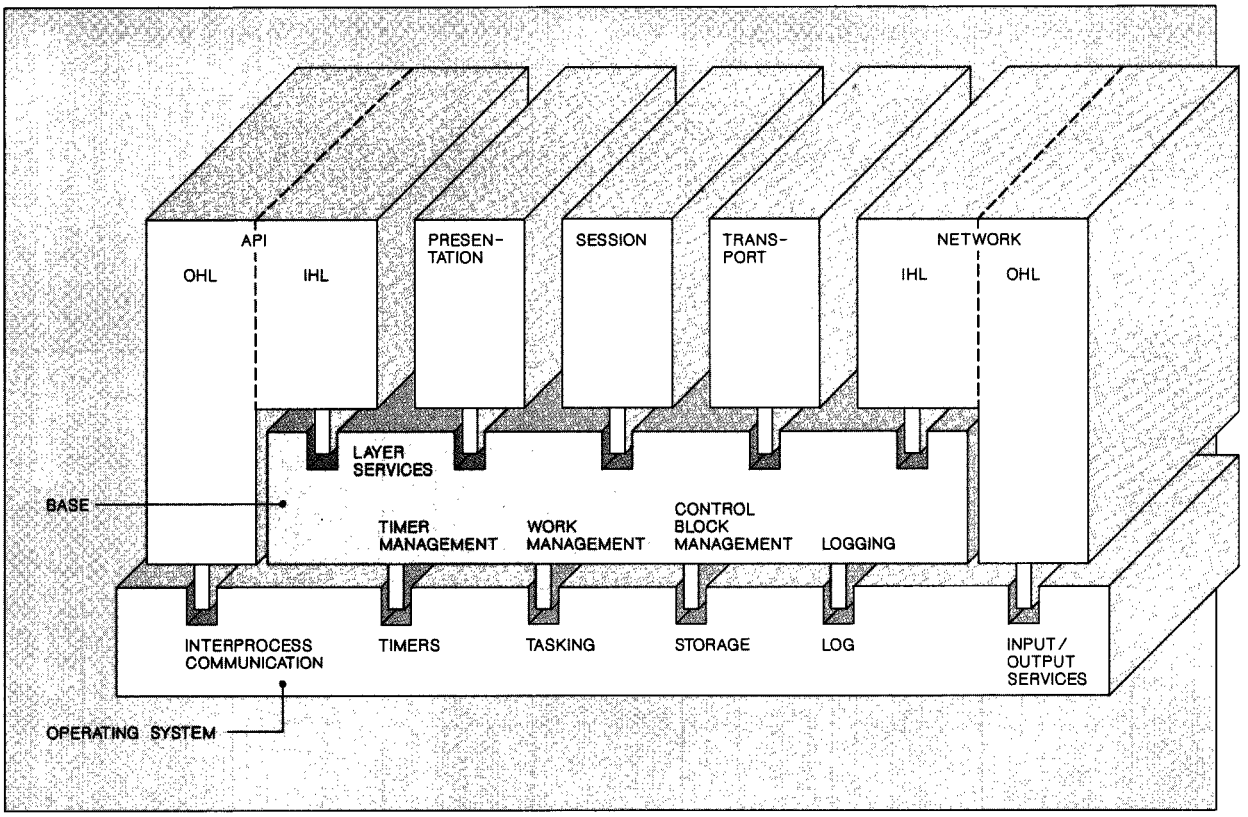
The OSI/Communications Subsystem will need access to the I/O and interprocess communication services provided by the operating system. To meet this requirement, one solution includes these services in the Base. Another, based on common operating system practice, separates basic services from services provided by modular "access methods." The latter is a far more general solution and was chosen for the OSI/Communications Subsystem. This approach allows Base implementers to concentrate on operating system services and not have to deal with the complexity of interfacing to network or LAN services. Developers with communications experience can be assigned to implement these communications-oriented components of each system.

The OSI/Communications Subsystem architecture includes a mechanism for adding system-dependent components to provide a direct interface to system services not already supplied by the Base. This mechanism supports a special type of layer known as a *boundary layer*. As the name implies, a boundary layer sits on the boundary between the environment established by the OSI/Communications Subsystem Base and the environment established by the underlying operating system and access methods. The part of the boundary layer inside the Base environment (*inner half layer*, or IHL) behaves as a regular layer, whereas the part outside the Base environment (*outer half layer*, or OHL) uses operating system services as needed. Boundary layer support in the Base includes a system-independent interface mechanism supporting communication between IHL and OHL. IHLs are provided with all of the same services as other layers. OHLs may use a subset of Base services and have additional special services available to them.

Continuing the previous analogy, we can consider a boundary layer as an "interface card" connected to the "motherboard." Part of the interface card extends outside the Base to interact with the "real" world while the other part of the interface card is controlled by the Base as a "processing card" (see Figure 2).

The OSI/Communications Subsystem uses this capability to implement three interfaces to other

Figure 2 Plugging boundary layers into the Base



processes in the system and to communications links:

1. The *Open System Manager (OSM) boundary layer* provides management services to all other layers within the subsystem and interfaces to system management processes such as NetView®.
2. The *application programming interface (API) boundary layer* enables the programming interfaces between the user applications and the communications subsystem, using the flexibility provided by the Base to interface with two different layers and provide both an ACSE/Presentation and a Session interface.
3. The *network boundary layer* interfaces with network access methods in a system-dependent way.

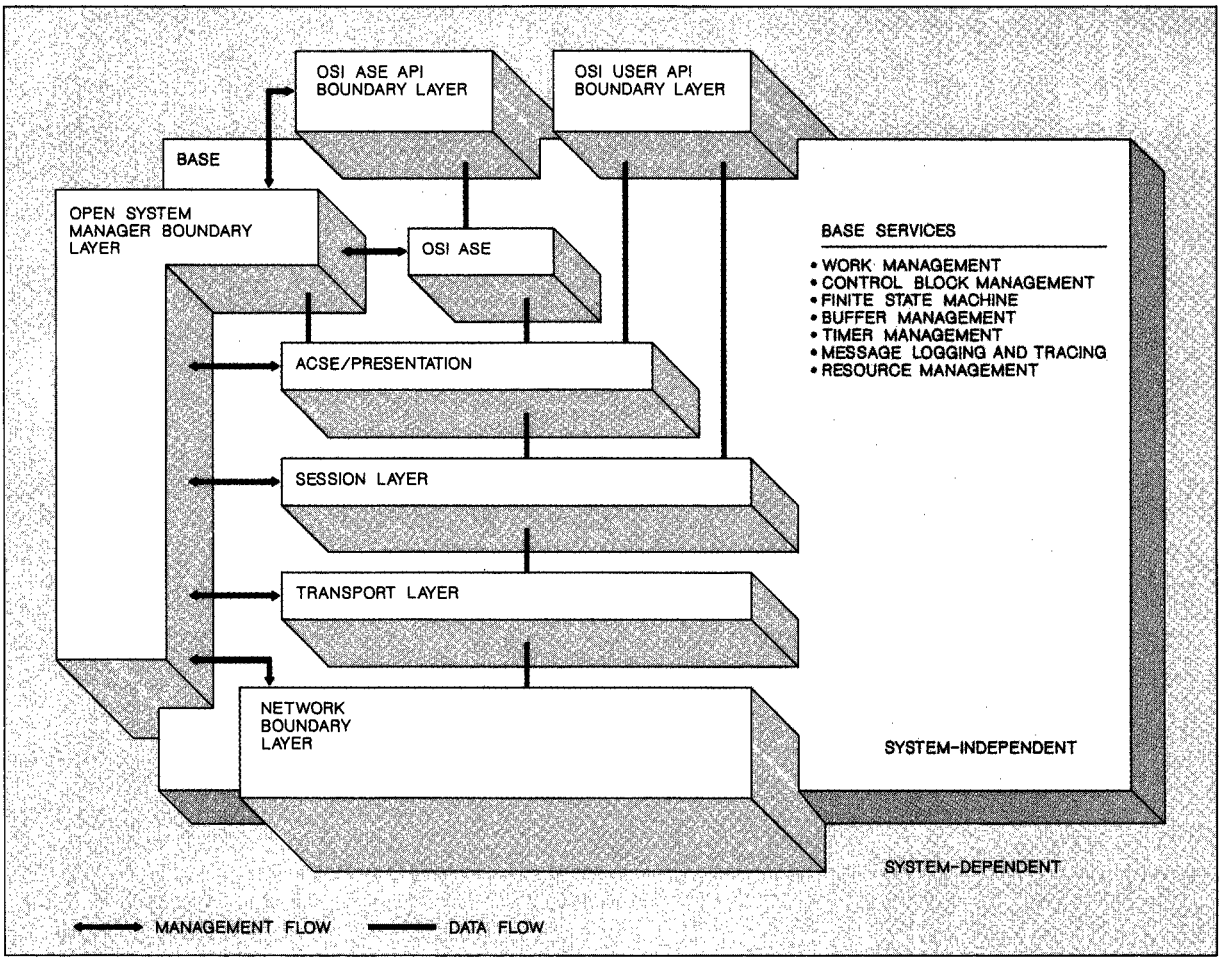
It is possible to implement additional OSI standards as a combination of one or more layers or boundary layers. This implementation is done, for example, by companion products to the OSI/Communications Subsystem: *OSI/File Services* implements the File Transfer, Access, and Management (FTAM) Application Service Element (ASE) of OSI, and *Open Network Distribution Services* implements the International Telegraph and Telephone Consultative Committee (CCITT) X.400 Message Handling System. These products are represented by the ASE depicted in Figure 3.

Figure 3 illustrates the Base and layers in Figure 2 from a perspective looking down onto the "tops" of the layers to show the relationships among them.

The categories of services provided by the Base are:

Control block management—Manages control blocks representing the OSI concepts of (N)-entity, Service Access Point, and Connection

Figure 3 Basic system structure –"top view"



End Point. Both storage and queues associated with these control blocks are managed by the Base.

Finite state machine services—Performs finite state machine transitions for the layers.

Work management—Includes a process model flexible enough to run effectively on a large range of processors (personal computers to multiprocessor main frames) with a built-in back pressure mechanism.

Buffer management—Provides a physical buffer management function across layers.

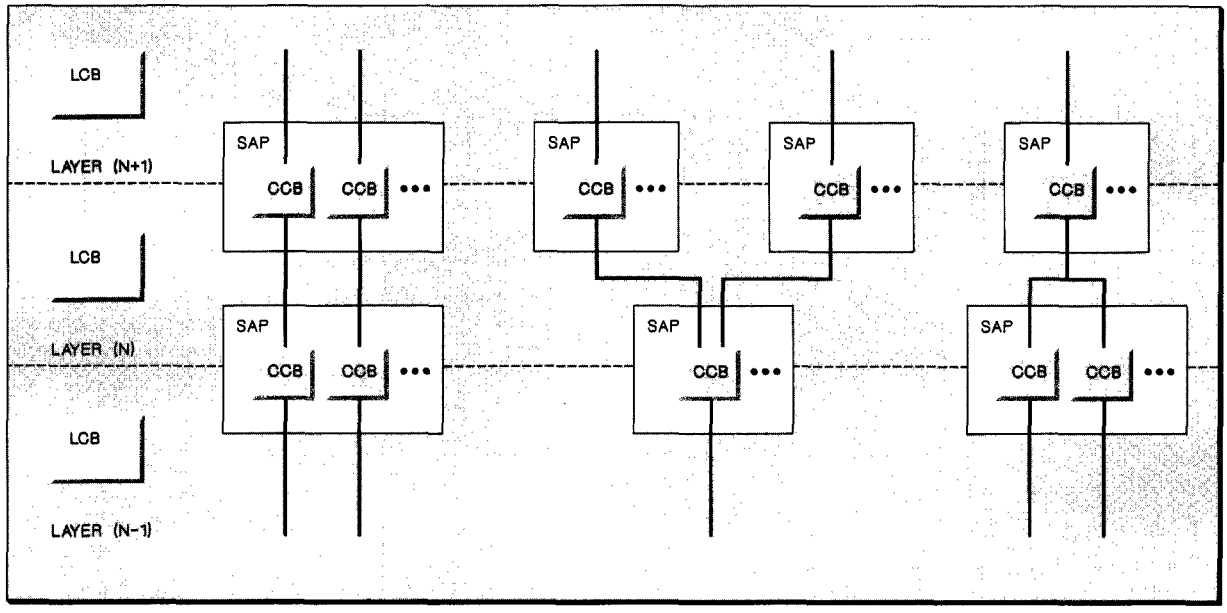
Resource management—Provides early warning of low storage conditions.

Timer management—Provides timer interrupts required by OSI protocols.

Message logging and tracing—Provides logging and tracing services required by communications products.

Control block management. There are three key concepts from the OSI Reference Model which are represented by the OSI/Communications Subsystem as control blocks:

Figure 4 LCBs, SAPs, and CCBs

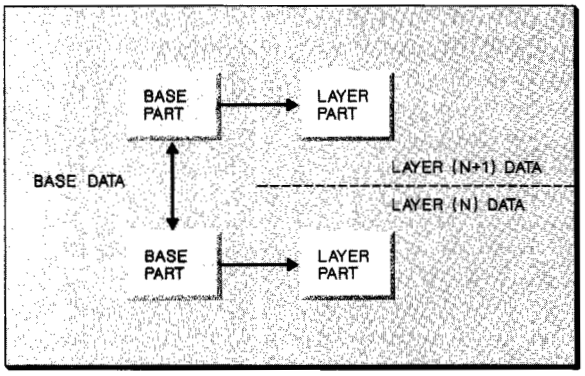


1. *(N)-entity*—an instance of a protocol machine of some OSI layer (Layer N) within the subsystem. For example, a Transport protocol machine in a subsystem is a Transport entity. Layers supported by the Base are like (N)-entities; however, several OSI (N)-entities may be implemented in one layer of the OSI/Communications Subsystem, and non-OSI components may be implemented in layers. Layers are represented by layer-entity control blocks (LCBs).
2. *Service Access Point* (SAP)—the point at which services are provided by an (N)-entity to an (N+1)-entity. The SAP defines the configuration of the system in terms of service-provider to service-user relationships between layers. It also defines the addresses of service-user (N+1)-entities within the OSI environment. In the OSI/Communications Subsystem, SAP control blocks are used to represent OSI SAPs, as well as other internal service-user to service-provider relationships between OSI/Communications Subsystem layers.
3. *Connection End Point* (CEP)—the representation of one end of a connection between two peer service-user (N)-entities within the OSI environment. Connections are provided as part of the service associated with a SAP, so each CEP is related to a SAP. (Connectionless OSI services do not use CEPs.) OSI/Communications Subsystem Connection Control Blocks (CCBs) represent CEPs.

The control blocks are most often used to represent these concepts, but it should be noted that there are no restrictions. It is possible for these control blocks to represent other OSI concepts or to even represent concepts not found in OSI. Figure 4 shows how the various control blocks relate conceptually to layers in a normal subsystem.

In general, an OSI layer performs services for the layer above it (a service-user) by using the services of the layer below it (a service-provider). (In Figure 4, Layer (N+1) is the service user of Layer (N), and Layer (N−1) is its service provider.) This method usually implies a relationship between CCBs to the service-users of a layer above and CCBs to service-providers below. For example, the Transport Layer supports its service-users by providing transport connections to them and uses network connections of the layer below. The Transport protocol implies some relationship between the transport connections above and the network connections below. Such a relationship may be simple and direct (data flowing in on one CCB always flows through to the other CCB) or more complex, involving multiplexing or splitting

Figure 5 CCB/SAP control block structure



protocols or even combinations of both. Some examples are depicted in Figure 4. CCB relationships may be reflected in *ties* between user and provider CCBs. The Base provides services that allow a layer to tie CCBs together, untie them, and locate tied CCBs quickly. Besides the layer use of this tie information, the Base uses it in its dispatching algorithms as discussed later in the subsection on work management.

The OSI/Communications Subsystem architecture implements strong data isolation required by software engineering. The control blocks that hold the system together are designed to ensure that each component has access only to the data necessary to its operation and cannot access or change data of other components directly.

The existence of the LCB is known by the Base and the layer associated with it. SAPs and CCBs define relationships between pairs of layers, and their existence is known to both layers in the relationship. Each control block is split into separate parts, which are conceptually and sometimes physically isolated. Each has a Base part and one layer part for each associated layer. A layer is only given access to its part of a control block and is not allowed to examine or modify parts belonging to the Base or to another layer. Although the Base allocates space and knows the whereabouts of each layer part of a control block, it does not examine or depend on the contents of any layer part. The Base part is used by the Base for managing the layer or relationship between layers. The layer parts of all of the control blocks associated with one layer make up the majority of the storage area that is addressable by that layer. A layer part is typically used to store state information that pertains to the object (such as a connection) that the control block represents. Layer parts of LCBs generally contain more static configuration data pertaining to the operation of the whole layer.

Enforcement of data isolation varies from one implementation of the Base to another, depending on the facilities available in a host operating system and the cost of using them. In some cases, all control block parts are stored in contiguous storage and are protected from access by nonassociated layers only because these layers do not know the addresses or the structure of the parts of other layers. Where a system provides protected data spaces, the structure allows the Base to take advantage of different data spaces when appropriate. For example, there could be unique data spaces for each layer and the Base without any impact to the common layer code (see Figure 5). Since this design prohibits layers from sharing data spaces, it isolates each layer from all of the others. Such layers can be packaged as part of a single processor system or be a part of systems distributed across multiple processors (see the subsection on future considerations for possible implications).

Internally, each layer part of a control block is assigned a unique control block identifier by the Base, and the entire control block is given a single external name. The Base provides services to manage control blocks: to create and delete them, to locate them by name or by identifier, and to retrieve information about them.

Finite state machine. The components that operate within the Base environment (layers and inner half layers) are designed to be event-driven; that is, they are invoked to process events. Their design is often based on finite state machine concepts. In a finite state machine, an event combined with the current state (remembered from a previous invocation) determines the action to be performed and the resulting state. The relationship of states, actions, and events are usually defined by a *state table*. OSI protocols are specified using state tables, so it is natural to implement them as finite state machines.

Finite state machine design tends to break complex functions into small, simple parts, and the state ta-

ble design forces consideration of all possible combinations of state and input. The performance characteristics of finite state machine designs are good, since a well-designed machine can select the appropriate action routine quickly.[8]

The Base provides a finite state machine service to layers. Invoked with an event, the current state, and a layer-defined state table, this service efficiently finds and invokes the appropriate layer routines and returns the next state. An event in this environment is a request to perform a particular function, together with the parameters and buffered data needed to perform it.

The Base also provides the layers with a good environment for finite state machines. Layers are invoked with a single event, given a single control block to which this event applies. There are no Base services that allow layers to wait for any event without returning to the Base, so layers must preserve state information between invocations. A typical layer when invoked performs an action (which may or may not generate additional events), places itself into an ending state (which may or may not be different from the entry state), and returns to the Base. A layer normally saves its state information in the layer part of control blocks. This action allows each layer, SAP, and connection to maintain its own (different) state.

Work management. Work management consists of three major parts: process model, flow control, and error handling.

Process model. A layer is invoked by the Base via a subroutine call to process an event for a particular control block. The event, control block identifier, and related data are contained in a *work request*. Work requests are issued by layers, boundary layers, or the Base, and each work request is directed to a layer, for a control block, with an event. Inside the Base environment, a *thread* performs much the same role as a task or process: It is the means by which the Base allocates processing cycles to layers. Work requests are associated with threads to schedule layer execution. Several may be associated with a thread in a last-in, first-out relationship. When a thread is dispatched, the work request most recently associated with it is performed by invoking the specified layer. Two different threads are considered to be asynchronous—and may be dispatched concurrently in Bases implemented on multitasking systems. As work requests are performed, their sequence of arrival on a control block is maintained.

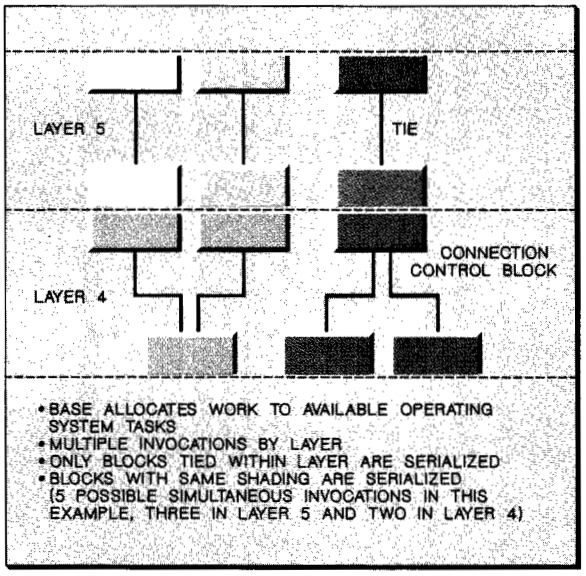
Since one layer normally uses the services of another layer, a protocol operation in the OSI/Communications Subsystem proceeds up or down through the stack of layers, each layer issuing work requests to its service-user or service-provider and using the same thread for the new work requests. Thus, each operation results in a sequence of related work requests associated with the same thread. When a layer issues a new work request on a thread, it may choose to leave the work request that invoked it on the thread or to remove it. Such a work request becomes a *recall* work request when left on the thread, because it will cause the layer to be invoked again after all processing related to the new work request has completed. A layer may also choose to issue work requests associated with a different thread to perform services asynchronous to the main flow.

Inside the Base environment, a thread performs much the same role as a task or process.

The Base uses the processing cycles allocated to the OSI/Communications Subsystem by the host operating system. One or more operating system tasks or processes may be available for use by the Base (depending on the Base implementation). The Base controls the dispatching of its threads by allocating threads to available operating system tasks. If it has only one operating system task at its disposal, it will alternate between the various threads and perform one work request at a time. However, if more than one task is available, the Base allocates active threads to the available tasks for the duration of one or more work requests, allowing a number of work requests to be performed concurrently.

If multiple tasks are available, the possibility of a collision (two different threads executing at the

Figure 6 Multitasking example



same time with the same control block) arises. For example, two protocol operations for the same connection may be flowing in opposite directions and meet at the same layer. In that case, the Base will serialize the requests for the layer in order to prevent multiple invocations for the same control block. When multiple CCBs are tied together within a layer, they are serialized as a unit. See Figure 6. For unrelated connections or SAPs, two threads are allowed to execute within the same layer concurrently, and threads may execute in different layers without restriction.

More traditional implementations of multitasking in a layered communications subsystem might allocate a task to each layer or a task to each connection.

The first solution imposes unnecessary task switching as data flows from layer to layer. In many systems, task switches are too costly for good performance in a communications subsystem. The second solution does not work well for OSI when multiplexing and splitting protocols are in use, since one connection becomes several connections at some point in the stack. This approach may require more tasks than are available in smaller operating systems. For multiprocessor systems such as the System/390, one task at a time per connection may not make the best use of system resources.

The Base work management approach overcomes these limitations, and does not fix the number of tasks for each implementation. While allowing implementations to assign tasks as above, it also supports single-task-per-system operation, use of invoking tasks, or use of several tasks in parallel. Multiple threads may be executing concurrently for the same connection, and multiple threads may be executing concurrently in the same layer (except for collisions as described above). Note that each implementation of the Base does fix its policy for assigning tasks—this flexibility is given to Base implementers rather than Base users.

Control blocks also have a role in work management; each SAP or CCB can be thought of as a work queue. When there is work to do on a connection, the work request is queued to the control block, and some task will come along and perform it. It may be, but does not have to be, the same task each time. Because of the finite state nature of the layer code, layers do not require dedicated tasks, and the Base model allows whatever tasks are available to float to any pending work requests. Once the Base has allocated a thread to a task, however, the task continues to perform work requests associated with the thread until there are no more or serialization contention is encountered. This allows work to proceed through several layers with minimal queuing overhead.

An additional feature of this approach is that, on those operating systems where it makes sense (such as OS/2), the Base allows user tasks to be borrowed to dispatch Base threads, avoiding a task switch between the user and the OSI/Communications Subsystem. In this case a user task may process related work requests as long as processing can continue without being suspended. Once suspended, the user task returns and processing continues when possible using a Base task.

The Base provides services that allow layers to manipulate work requests and threads. One important service is the ability to *suspend* a thread. When work cannot continue for a thread (because of protocol flow control restrictions or the need to perform some I/O operation, for example), a layer can cause the thread to be suspended as it returns control to the Base. Such a thread is no longer

dispatchable and is enqueued on the control block of its topmost work request. The layer can cause it to be *resumed* when performing a different work request, normally one that frees flow control restrictions or indicates completion of I/O operations. The thread becomes dispatchable again, and work requests associated with it are performed when it is next dispatched.

Another service is the ability to *hold* a work request. This service effectively removes a work request from a thread and enqueues it to its control block in such a way that the layer can refer to the associated data through several invocations. Held work requests become static data areas for use only by the holding layer. A layer can cause a held work request to be redispatched at a later time if necessary by placing it on an available thread.

A layer or a Base service may detect a condition that should not happen.

Layers usually process work requests once before holding them. For example, the Transport Layer may have to guarantee delivery. After it sends out a message, it will hold the work request containing the message until an acknowledgment is received. If the acknowledgment is not received, the work request will be rescheduled. If the acknowledgment is received, the work request will be deleted.

Flow control. One challenge a communications subsystem faces is the ability to manage its data flow requirements within available data storage resources. This challenge cannot be satisfied unless the communications protocols contain flow control functions. In OSI, the Transport Layer and the Network Layer share responsibility for flow control protocol. However, these functions alone are not sufficient unless their effects are reflected back to the local applications sending and receiving data. For instance, if the Transport Layer acknowledges data upon receipt but before passing it to the application, the local subsystem has defeated the Transport Layer flow control function and cannot control the amount of data to buffer pending receipt by the application. If the application may queue data to send without controls, an application may send data faster than it is transmitted and may exhaust the available resources to buffer the data prior to transmission.

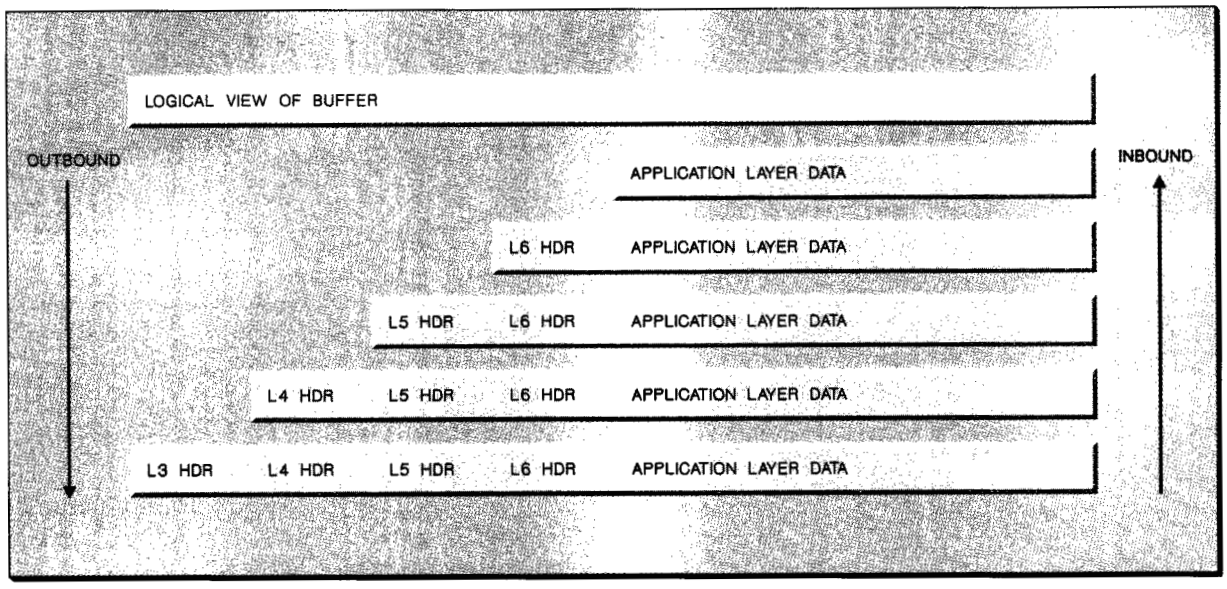
The OSI/Communications Subsystem has chosen a solution that extends the available protocol flow control function throughout the entire communications subsystem by relating it to the basic threading mechanism. Feedback between users and flow-control protocol is done by using a feature of the Base thread and work request mechanisms that allows more than one work request on a thread at a time. The thread acts like a stack, where the bottom work request will not be performed until all above it have been performed. Since new work requests are placed on top of the stack, they are performed before the bottom one.

Normally, work requests are removed from a thread after they are performed, but a layer may choose to leave a recall work request on the thread while adding a new work request above it. Layers supporting a source of data such as a user or the network can become aware of flow control restrictions by using the recall function. Layers needing to control the flow of data for any reason can control the timing of recall invocations of source layers by suspending threads until they are ready for more data. Intermediate layers are not aware of this function (they do not see the recall work requests) but have a responsibility to ensure that the same thread is used to pass work through. The recall signals the originator that more data may be sent. This mechanism works without requiring any special functions in layer interfaces and without affecting intermediate layers.

Error handling. There are cases when a layer or a Base service may detect a condition that should not happen. This detection is considered an error in the software. There is a Base service to abnormally terminate (*abend*) the execution of a thread in these situations. This service also provides a single point for collection of the information necessary to identify and correct such errors.

This abend service allows the layer to end the current invocation immediately. It simplifies the code within a layer since much return code check-

Figure 7 Layer view of buffer



ing and passing may be avoided within the various subroutines of a layer. In particular, a Base service detecting a layer error will issue the abend service directly and not even return to the layer. For those situations where the Base cannot determine whether the layer is in error, the Base will invoke a special layer subroutine to separate unexpected errors from others. Base services only pass back return codes that the invoker is expecting to handle. Layers need not have code to deal with unexpected results after each Base service.

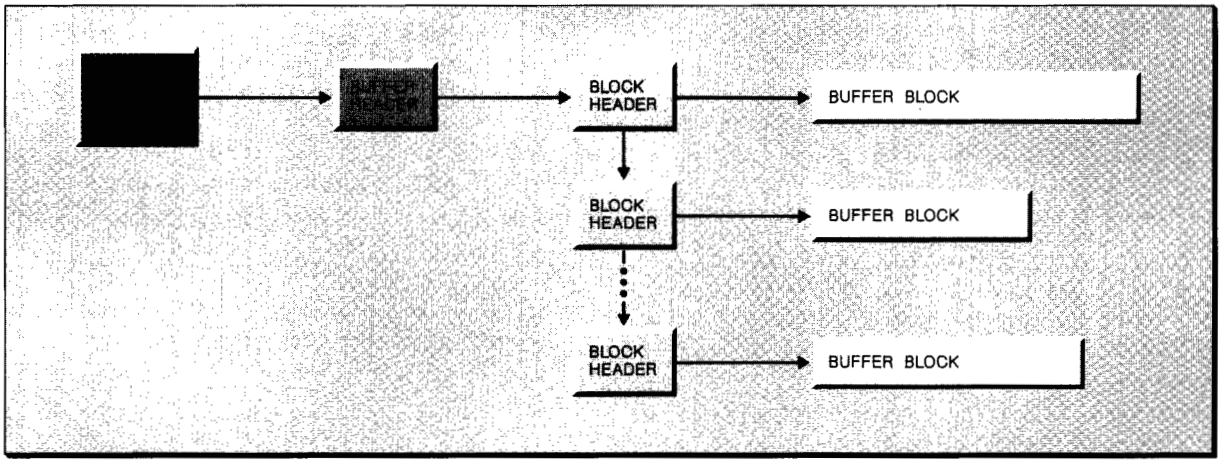
Buffer management. The Base provides buffer management services to the layers. These services allow the layers to treat a buffer as a contiguous byte-stream, without regard to the actual physical location or makeup of the buffer. Layers use these services to examine and modify the contents of buffers. Direct addressing to the contents of buffers is not normally provided. This scheme allows the Base complete freedom to locate buffers in small address-space machines. It also allows a layer to perform logical, rather than physical, operations on a buffer. For instance, a layer may add a header to the contents of a buffer without worrying about whether enough physical space exists at the front for the header. This logical view is depicted in Figure 7.

The internal model of a Base buffer consists of a buffer header and a chain of separate data blocks, each with its own separate header record. Each data block is composed of a physically contiguous stream of bytes but is not necessarily contiguous to the next data block or the same size. To minimize page references, it is useful to view the buffer and block headers as physically separated from the data blocks. Each buffer includes the data blocks needed to hold the buffer contents. This physical view is depicted in Figure 8.

Layers normally view buffers as logically contiguous streams of bytes and are not aware of their physical view. However, there is a buffer service that allows layers to get the direct address of a segment. Use of the direct access service allows layers to improve buffer access performance for certain types of operations, such as repeatedly accessing large data records saved in a buffer for local use or examining all data octets in order to generate a checksum. This service is not meant to be used during normal data transfer when only headers are being added or deleted to the buffer as the data pass through the layers.

Boundary layers may access buffer blocks to perform I/O operations directly into them. Special

Figure 8 Model of Base buffer



services available to outer half layers provide access for this purpose.

Buffers may be passed from one layer to another by passing an identifier to the buffer without making a copy of the buffer. When a layer adds a protocol header, a block containing the header may be added to the buffer. There is no need to move the data within the buffer to make room for the header.

Resource management. The OSI/Communications Subsystem environment makes it possible to set an upper bound to use of resources based on the number of SAPs and connections supported. The flow control mechanism allows layers managing sources or receivers of data to control the number of buffers that may be outstanding at one time. Rules for intermediate layers require them to set upper bounds on the number of resources used. When sufficient storage is available, storage depletion should not occur. However, having sufficient storage requires allocation of much storage that will never be used in real operation, and in practice it is not often feasible. In addition, certain protocols do not include a flow control function and have no reasonable means to control incoming data (for example, Connectionless Network Protocol when used as a relay). For this reason the Base provides additional resource management services.

The Base provides a service to aid the layers in the proper utilization of system resources. The only resource presently monitored is system storage. System storage is obtained by the Base for buffers, control blocks, and work requests. Should system storage become exhausted, normal operations would be impossible, and recovery would be quite difficult. Rather than trying to recover from this condition, the philosophy is to use early detection to prevent this condition from happening.

The Base monitors available storage and maintains a storage condition state. The storage state would change as the percentage of storage used crosses certain preset values. The exact values used could be an installation parameter, or they could be determined some other way, depending on the environment. To prevent frequent fluctuations, a different deadband setting would be used when returning to a previous state. For instance, consider states such as All Clear, First Warning, and Final Warning with the following state change settings:

- All Clear to First Warning at 70 percent
- First Warning to Final Warning at 90 percent
- Final Warning to First Warning at 80 percent
- First Warning to All Clear at 60 percent

A layer may use a Base service to be notified when a particular storage condition state is en-

countered. Notification consists of a normal layer invocation using a work request setup by the requesting layer when issuing the notification service. The action taken by a layer depends on the particular layer. Possible actions to lower storage requirements include limiting new connections, limiting incoming data, and reducing window sizes at the Transport Layer.

Timer management. The Base provides timer services to the layers. A layer may request that timer services trigger an event (by performing a work request provided by the layer) on a control block after a specified time period and may cancel outstanding requests if they are no longer needed. If the time period expires while the timer request is still outstanding, the Base schedules the work request under a new thread of execution to the layer. The work request indicates that a timer expired and is associated with the control block specified by the layer. When invoked after a time out, the layer may do whatever processing is necessary, including starting or resuming other threads.

The Base also provides time-of-day information in the various forms specified by the OSI standards.

Message logging and tracing. The Base provides services to aid in logging messages and in tracing activity within the communications subsystem. The services provided are standard for this type of activity. The system-dependent portion of the Base uses the logging and tracing services of the local operating system in the best way to satisfy the normal operating system procedures.

Message services support the separation of message text from the program code to allow for easy translation. Layer code may supply filler information in order to add dynamically determined data to the message. A layer specifies the type of filler data and an associated field number. The message text indicates where in the message a field is to be placed and how to format it for display.

On command, the Base will trace a layer invocation, including all of the pertinent information associated with the work request. Tracing may be designated for an individual control block, a group of control blocks, or for all control blocks. When errors are detected, other services are available to trace additional information determined by the individual layer.

Layer developers' view. To clarify the nature of the Base environment, it helps to see it from the perspective of its users—developers of portable layers.

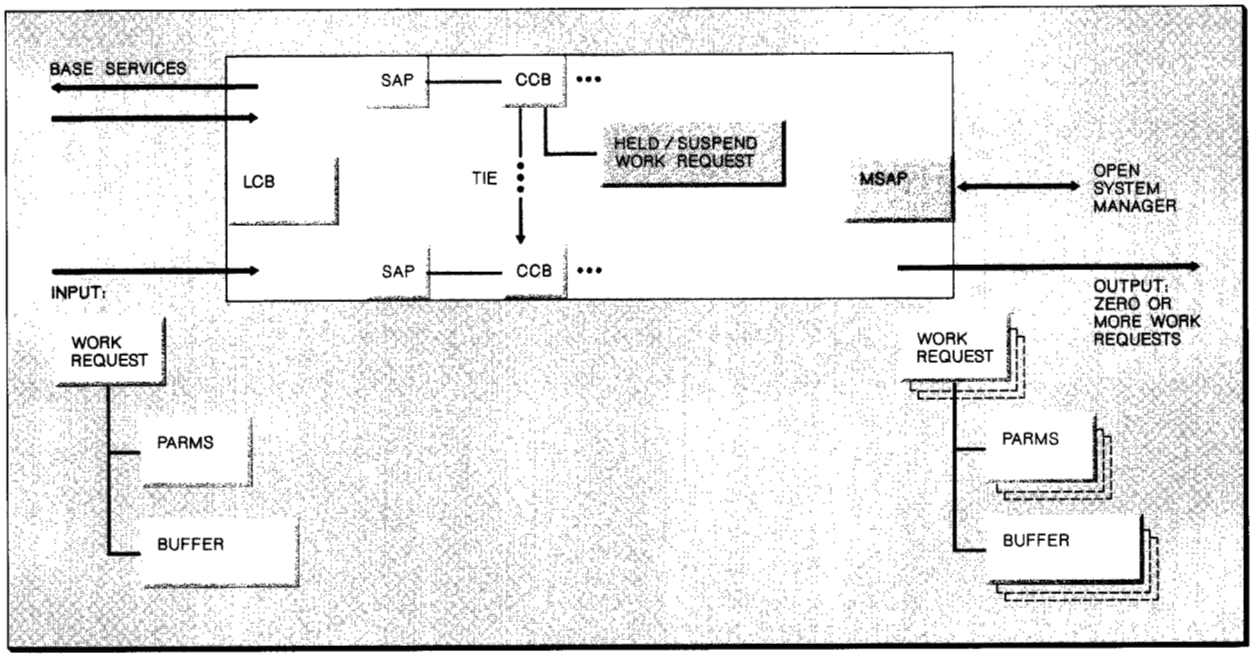
Before any code can be ported, it must be written in a programming language supported by the target systems. The OSI/Communications Subsystem has chosen Pascal. In comparison to many languages, Pascal is an easy-to-learn, highly structured language that produces reliable code and is widely available in current systems. However, several languages today meet this requirement. It is not the intent of this paper to address the possible choices or advantages of one language over another. It is important to choose some programming language that has a reasonable degree of standardization across different systems.

Usage of the chosen language must be further controlled through coding guidelines. Even in standardized languages, there are features that do not port well, and programmers must be restrained from using them. In the Base environment, Pascal is used only for generating in-line code; all run-time support is provided by the Base. The guidelines used in the OSI/Communications Subsystem ensure that no Pascal run-time support is used.

Figure 9 shows the environment provided by the Base from the perspective of the developer of a layer. The environment has the following characteristics:

- The layer is a subroutine of the Base—a called Pascal procedure. It is called when there is a work request to process. When the layer has completed all of the work associated with the single work request, it returns to the Base.
- The operation of layers is transaction-oriented, that is, broken up into small atomic items of work. Each invocation of a layer should result in a relatively small operation, not a long process. (There is no check on layers to enforce this rule, however.)
- The layer has no way to access the data or programs of another layer. The addresses of other layers are not provided.

Figure 9 A layer's view of the Base



- The Base provides the only persistent working storage available to layers: LCB, SAPs, CCBs.
- Temporary working storage is also provided by the Base: held or suspended work requests with parameters and buffers.
- Base services provide the only interface to system services. Layers do not have direct access to system I/O operations or other services.
- Base work management functions provide the only interface to the other layers in a subsystem. When a layer is processing a work request, it may need to invoke the services of other layers. It handles this need by building work requests for the other layers and scheduling them, not by calling the other layers.
- Layers have a consistent interface to common management: the MSAP association to the Open System Manager (OSM). These services are invoked via work requests, just as those of any other layer. The MSAP is used to carry these work requests.

Implementation experiences

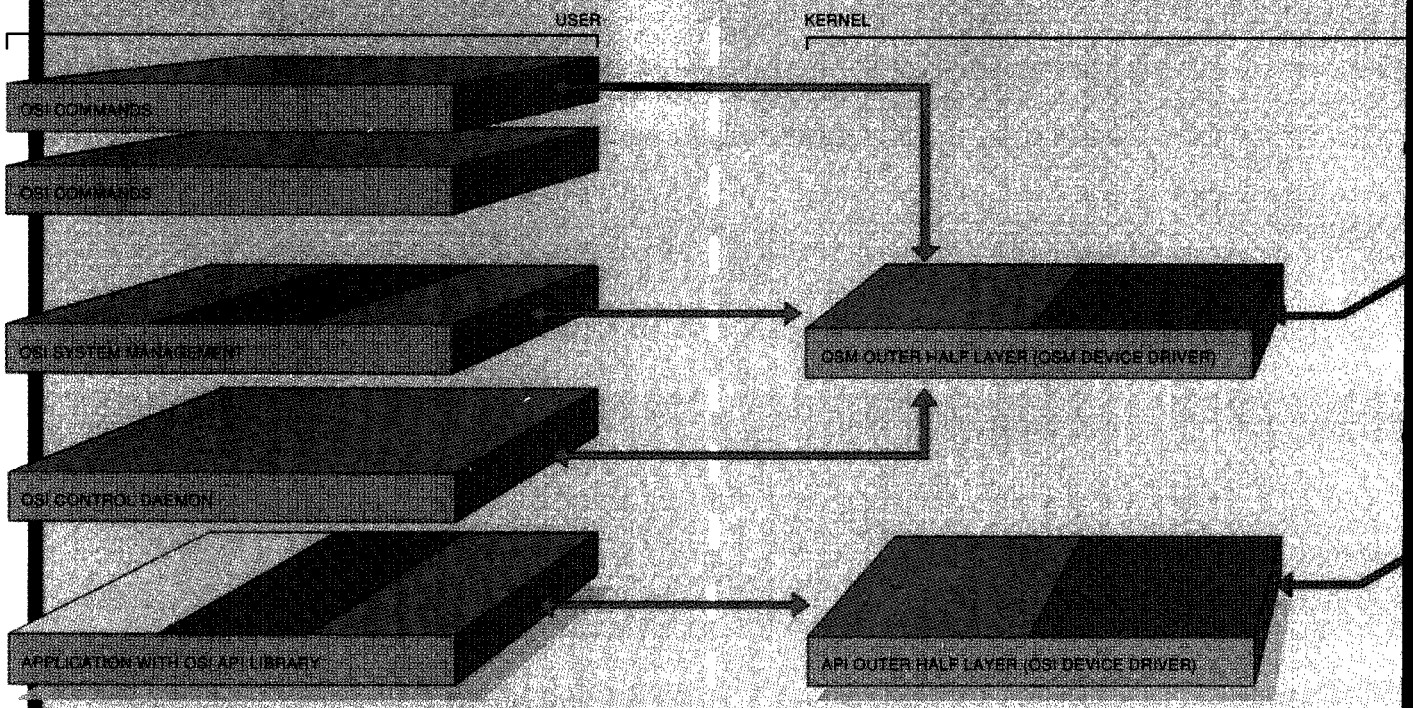
We have now had five years of experience developing systems based on this architecture. Among the results are:

- Early implementations on System/370™ MVS and Series/1 RPS were made available as limited customized offerings (known as programming requests for price quotation, or PRPQs) in 1987.
- Implementations on the SAA systems are available.
- Work to prove feasibility has been done to various stages of completion on the Advanced Interactive Executive™ (AIX®) and several other IBM systems and components.

Our experience with these systems has established that the Base can support successful implementations of OSI protocols. The layers of the OSI/Communications Subsystem are all portable with the exception of small modules containing system-specific constants and routines.

OSI/Communications Subsystem developers working in Pascal in the Base environment have achieved high productivity rates and excellent code quality. The Base environment is a relatively complex one for developers, however, and new developers require time to learn it before they can begin to be productive. Once it is familiar, programmers gain productivity because the Base takes care of many of the worrisome func-

Experiences in



The Open Systems Interconnection/Communications Subsystem (OSI/CS) was deliberately designed to minimize or eliminate problems found in "porting" a large system: language or compiler differences, system call semantic and syntactic differences, and dissimilar user interfaces. Outlined are experiences gained in experimentally porting the OSI/CS Base to AIX® Version 3. Although the work was not completed, porting the Base illustrates several key features of OSI/CS.

Layer and application code common—The main advantage of porting OSI/CS over writing an OSI system from scratch is that almost all of the layer and application code is common to all operating systems. The porting team need not even learn much about OSI; this code is simply provided, already running and tested for conformance.

System-dependent parts of Base confined to a few routines—An unexpected gift is that even the Base and outer half layers, which are considered system-dependent in the Base architecture, contain large amounts of common code. The routines that call system-dependent services are typically short and easy to identify. In a program not designed for easy porting, the system-dependent parts would be scattered throughout the entire program, requiring many large modules to be rewritten.

Standard Pascal used—Since the OSI/CS common code is intended to be portable to any operating system, it is written in a subset of Pascal that is standard to all systems. In a program not designed for easy porting, the programmers typically use nonstandard language features, which would have to be rewritten for a new system.

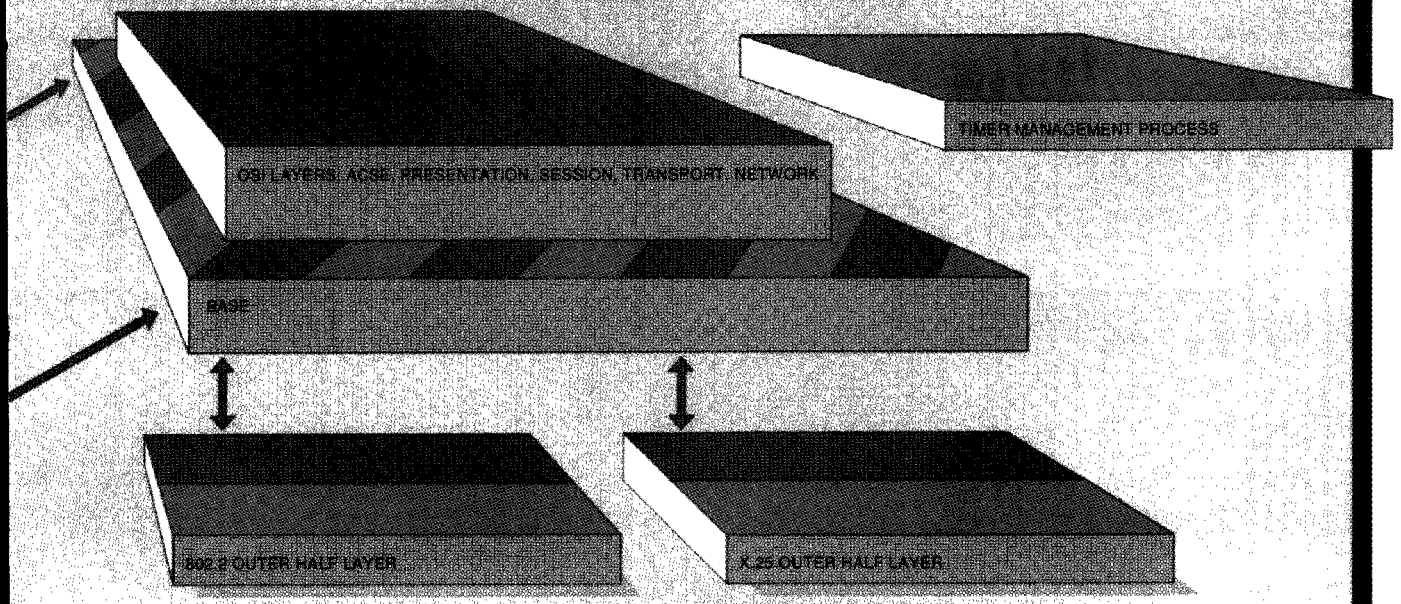
Isolation of user interface—The parts of a complete OSI/CS system that interface with the user are well isolated from the Base and layers, in outer half layers. This isolation simplifies the task of identifying and modifying system-dependent portions.

Other Bases available—Starting with a sample Base written entirely in Pascal, the AIX porting team was able simply to transfer this code to the new system and compile it there to get an initial Base implementation up and running. Several parts of the AIX Base were able to be used almost unchanged from the sample Base: the buffer management services, the finite state machine services, large parts of the work management services, as well as control block management services.

Since OSI/CS is a product that runs on several operating systems, the Base code, design documentation, and test layers already exist for these systems. The AIX porting team was able to use many of the timing services routines and memory allocation routines from the OS/2® version with only minor changes.

A few characteristics of the Base, listed below, make new porting a challenging undertaking. The overall job, however, is still much easier than writing a complete OSI system or than porting a system that was not designed to be ported.

Complexity of Base logic for multiprocessing support—A difficult part of porting the Base is simply understanding its complexities, particu-



larly the parts that are designed to support multiple threads of control and the serial use of the many control blocks. The porting team had to be sure that no problems were introduced when changing code in this area due to differences in tasking structure among the various operating systems. Because of the unpredictable nature of timing-dependent interrupts, this area was also very difficult to test.

Interlanguage calls—For various reasons, some parts of the Base should be written in assembler or a native system programming language rather than in Pascal. Among these reasons are: to obtain more efficiency in the very lowest level Base functions such as locking, to make system calls that require assembler language macros or C language header files, and to be able to use tables of layer branch points that require the type "pointer to function." Therefore, the porting team needed to solve the problems inherent in any two-language system: synchronizing changes to two versions of declarations of data, and accounting for language differences in the representation and alignment of data types.

Half-layer services and structure—The Base architecture for half-layer services assumes that communication between all outer half layers and their corresponding inner half layers can be done in the same way. In the porting to AIX, the network outer half layers were part of the kernel of the operating system, whereas the API and system management outer half layers were user applications. Thus a half-layer service needed to call one set of system services for network OHLs and a very different set for user-space OHLs.

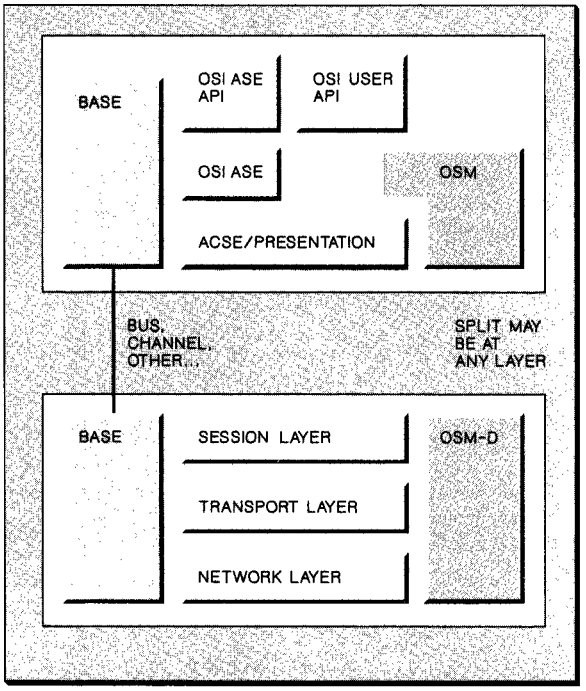
Resource management architecture—In order for the Base resource management scheme to work well, the subsystem must be assigned a fixed amount of memory over which it has control. In AIX, however, the Base used a system heap shared by other system routines such as device drivers. In this situation no easy way exists to prevent overrun of this heap, since the OSI subsystem has no control over the other users sharing the heap.

The accompanying figure shows the structure of the experimental OSI/CS within AIX. System-dependent parts are shown in green, and system-independent parts (unchanged) are shown in blue.

The experimental porting to AIX was accomplished in a location remote from the Base developers, without their full-time support. Out of the total amount of code needed for a complete OSI system including Base and layers, only about 10 percent had to be changed or newly written (this number does not include code for the outer half layers and the OSI applications, which were not attempted for AIX). Thus, for a relatively small amount of effort, and with almost no expertise in OSI protocols, a complete OSI system can be produced. Despite the difficulties encountered, the overall experience in the experimental porting of OSI/CS to AIX demonstrated that the design was well-structured, portable, and flexible. It resulted in a system that exploited the unique features and capabilities of AIX, took advantage of sophisticated communications techniques, and required a small amount of new code to be written.

Nancy Crowther, Joyce Graham

Figure 10 Overview of split OSI/Communications Subsystem architecture



tions needed for a communications product. The early maintenance on the SAA products indicates that the portability of the code has reduced cost. A problem found on one system and fixed there has provided a portable fix to the other systems, usually before it was reported by users.

The portable layers and other ported components in our product have resulted in a significant leverage on development cost. The products we have shipped included two and one half times more lines of code than our developers have produced and will maintain. Although this advantage is significant, it may be less than expected considering the size of the Base and relative size of the layers. The cost of building a subsystem into another operating system, so that it takes on the characteristics of the system environment, requires a significant amount of code that is specific to that system. Especially expensive are the OHL components that map system access methods and management functions into the standardized functions of the Base and layers.

Path length measurements in similar situations have indicated that the Base model can produce improvements when compared to task-per-layer implementations. Comparison to a predecessor OSI product on the System/370 (Open Systems Transport and Session Support) has shown that path lengths for the OSI/Communications Subsystem are significantly smaller than that of the old products for a sample configuration. However, experience shows that careful design is necessary to achieve good performance.

Placement of the Base within a system is especially important for performance. Since the Base and layers are portable and written in a high-level language, they would appear to be suitable for execution in an application environment. However, they still implement a communications subsystem and must have fast, direct access to system services in order to achieve their potential performance.

Future considerations. Many extensions to the Base may be possible in the future. One such possibility is the split OSI/Communications Subsystem. It allows multiple Bases to operate in a uniform manner so as to comprise a single OSI/Communications Subsystem. Figure 10 provides an overview of what such a system might be. The Base control block structure and process model allow this configuration to take place without an impact to most of the OSI/Communications Subsystem. Only the management layer (labeled OSM) and the Base would know about the split configuration. OSM would have a counterpart (labeled OSM-D) to help manage in a split configuration.

One possible use for a split OSI/Communications Subsystem would be to allow part of the OSI stack to reside within an operating system kernel and part outside. Another possible use might be to place part of the OSI stack within a front-end processor and part within the main processor.

Summary and conclusions

The Base architecture, as used in the IBM OSI/Communications Subsystem, provides a method of implementing portable layered OSI implementations. The Base provides services to support the special needs of communications protocols and allows porting of protocol machines across systems. It includes techniques to access system-dependent

functions of the underlying system. Special architectural support is provided for management, application interfaces, and connectivity interfaces into the portable protocol machines. Layer implementers are provided with an environment that isolates their component from the other layers in a subsystem and automatically ties state information to relevant events. Services allow representation of the complex relationships among connections and service access points at various layers.

The work management services of the Base are designed to give implementers flexibility in how system processes are used and mapped to Base functions. The buffer services eliminate movement of buffered data between layers.

The architecture has been used successfully to port implementations of OSI layers across several different systems with quite different internal architectures. The concepts can be extended in the future to provide support for these protocols in new environments, including those implemented in a distributed manner.

Acknowledgments

The general architecture of the Base and the OSI/Communications Subsystem was developed during 1985 by a team of system designers including the authors, Jerry Holten, and Rob Koning. We gratefully acknowledge the contributions of the other team members and those of others too numerous to list. We give special recognition to the drive and ideas of Giuseppe Facchetti.

Cited references and notes

1. ISO 7498, *Information Processing Systems—Open Systems Interconnection—Basic Reference Model*, International Organization for Standardization, Geneva (October 1984).
2. *IBM Programming Announcement 288-490* (September 20, 1988); available through IBM branch offices.
3. *System/370 MVS and VM OSI/Communications Subsystem General Information Manual*, GL23-0184, IBM Corporation; available through IBM branch offices.
4. D. D. Clark, "The Structuring of Systems Using Upcalls," *Proceedings of the 10th ACM SIGOPS Symposium on Operating Systems Principles*, Orcas Island, WA (December 1985), pp. 171–180.
5. H. R. Alberecht and L. C. Thomason, "I/O Facilities of the Distributed Processing Programming Executive (DPPX)," *IBM Systems Journal* **18**, No. 4, 526–546 (1979).
6. *UNIX System V Release 3 STREAMS Programmer's Guide*, AT&T, Summit, NJ (June 1986).
7. OSI/Communications Subsystem layers do not correspond to OSI layers. In OSI, a layer conceptually includes functions in all open systems; a subsystem is the part of a layer within one open system; and an (N)-entity is an implementation of layer functions (for layer N) within a subsystem. The OSI/Communications Subsystem layer corresponds most closely (but not exactly) to an OSI (N)-entity.
8. IBM Systems Network Architecture includes finite state machine components, and several IBM products (such as the 3745 Network Control Program) implement finite state machines.

General references

J. R. Aschenbrenner, "Open Systems Interconnection," *IBM Systems Journal* **25**, Nos. 3/4, 369–379 (1986).

A. Fleischmann, S. T. Chin, and W. Effelsberg, "Specification and Implementation of an ISO Session Layer," *IBM Systems Journal* **26**, No. 3, 255–275 (1987).

S. H. Goldberg, "Back-Pressure Flow Control Using a Thread Recall Mechanism," *IBM Research Disclosure N-306*, IBM Corporation, 5600 Cottle Road, San Jose, CA 95193 (October 1989).

L. Svobodova, "Implementing OSI Systems," *IEEE Journal of Selected Areas of Communications* **7**, No. 7, 1115–1130 (September 1989).

Steven H. Goldberg *IBM Communication Systems, 1501 California Avenue, Palo Alto, California 94304.* Mr. Goldberg is a senior programmer in the OSI Architecture Department in the Palo Alto Programming Center and lead architect for the Base component of the OSI/Communications Subsystem. He joined IBM at Poughkeepsie, New York, in 1967 and has had extensive experience developing IBM operating systems for System/360, System/7, and Series/1. In 1979 he transferred to his current location to work on communications products. He has an M.S. in computer science from Ohio State University.

Jerome A. Mouton, Jr. *IBM Communication Systems, 1501 California Avenue, Palo Alto, California 94304.* Mr. Mouton is a Senior Technical Staff Member and lead architect for OSI/Communications Subsystem products. He joined IBM in 1967 and has worked on a number of IBM products, including the DB/DC Data Dictionary, the SNA Network Performance Analyzer, Series/1 EDX Communications Facility, and the OSI/Communications Subsystem. Mr. Mouton received a B.A. in philosophy from Rice University in 1969.

Reprint Order No. G321-5434.