Chapter 2.0 The definition of x-kernel and OSI

The x-kernel is a network-based software which implements network based protocols. It is a new environment for implementing and composing network protocols. This x-kernel was basically built as an attempt to improve the existing environment of implementing and testing network protocols [3]. Examples of famous network protocols implementation environment to date are, Berkeley's (BSD) and System V, which uses the Unix operating system as its platform. The x-kernel does what normally the protocol's service interface and peer-to-peer interface does, which is to communicate between protocol layers and peers.

2.1 The OSI Model

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. Figure 2.1 depicts an OSI model of network architecture. 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. Figure 2.2 depicts the service interface between protocols and peers. The familiar layered diagram is misleading since it only depicts a layered monolithic structure and omits the peer-to-peer aspects of the model. 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. The OSI standards define the service model and specify the details of the protocol for each layer, but leave most implementation issues undefined.

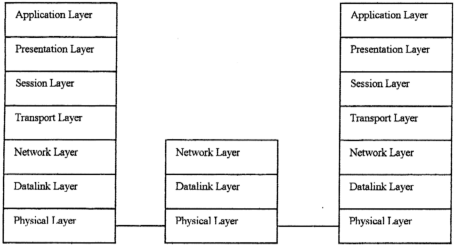

Figure 2.1 The OSI network architecture [1].

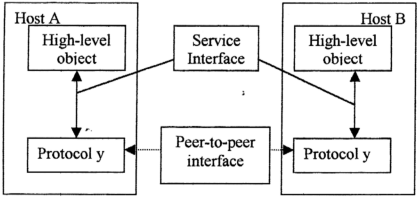

Figure 2.2 Service Interface between protocols and peers [3].

2.2 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 Remote Procedure Call (RPC) mechanism. The usefulness of the RPC paradigm in supporting client-server inter-actions is well known and is not discussed. In this section, the requirements for the OSI upper layers are presented. Figure 2.3 illustrates a common multi-protocol architecture used to build distributed object-oriented applications in Unix workstation environments.

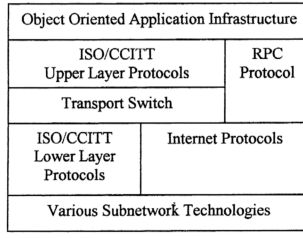

Figure 2.3 Unix-Based Multi-Protocol Architecture [7].

The transport switch layer depicted in Figure 2.3 is not part of the standard OSI model. The switch addresses, in an elegant manner, the pragmatic issue of simultaneously using many

underlying transport services arising from different transport/network protocol combinations. Different protocol combinations are required due to the varying characteristics of different subnetwork technologies. 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. 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, which expects a reliable connection-oriented network service. Thus, using the convergence 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.

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. 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. 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. Research on lower layer protocol implementation has led to programming techniques for stream-lining control flow within a process. From a methodology perspective, the most important contribution to network programming to date is the idea of structuring protocol implementations in terms of upcalls [17]. The upcall methodology requires that the implementation language support higher-order functions; 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

A protocol provides a communication service that higher-level objects such as application processes or higher-level protocols use to exchange messages [1]. Each protocol or layer defines two different interfaces. First, it defines a service interface to the other objects on the same computer that wants to use its communication services. This service interface defines the operations that local objects can perform on the protocol. Second, a protocol defines a peer interface to its counterpart on other machine. This second interface defines the form and meaning of messages exchanged between protocol peers to implement the communication service.

The x-kernel is poised as an Application Programmer Interface (API) because it supports the rapid implementation of network protocols. A protocol implementation can be done rapidly because the x-kernel provides a set of high-level abstractions that are tailored

specifically to support protocol implementations [2][3]. The x-kernel protocols are efficient because these abstractions are themselves implemented using highly optimized data structures and algorithms. The API with the support of object abstractions and routines are specially built or designed to implement network protocols in a minimal object oriented manner. The programming language that is used to implement the protocols is the classic ANSI C programming language This language is suited with the subtlest effects of object oriented programming to that of the C++ programming language to code network protocols. The x-kernel is not merely an object-oriented software but is built on the concepts of object-oriented programming [4]. The C programming language is used to code structures into libraries and routines; and the structures that share the common properties are put into the same type of classes. A standard set of abstractions in the x-kernel is grouped together to form a uniform interface to implement protocol abstractions. This interface is the UPI (Uniform Protocol Interface), this interface is the core for the x-kernel to build the syntax for communication and set the parameters for the network protocols. A deep understanding of interfaces provided by the x-kernel helps to understand the intricacies of the mechanism in network protocols.

The x-kernel includes components that manage processes, memory and communication. These components are supported by special routines in the x-kernel. Objects in the x-kernel are data structures, with a collection of operations that can be exported, to other objects class. The x-kernel gives a strong representation of the service interface between protocol layers, its infrastructure takes care of the invoking procedures that implement particular operations on two main abstractions which is the protocol object and the session object. With the support of the message library, participant library event

library and map library enables the x-kernel to configure and implement protocols and protocol graphs such as IP for the former and TCP/IP for the latter. Figure 2.4 shows a typical protocol stack of the TCP/IP protocol graph. In the figure protocols such as IP or TCP represent protocols objects, and the session object represents the local endpoint of the particular channel opened. This session objects are dynamically created as channels are opened and closed. Channel object is an interpretation of a respective protocol object. Channel objects are created as protocol objects export operations for opening channels. Channel objects exports operations for sending and receiving messages with the use of the routines xPush and xPop respectively. Exports operations of objects in the protocol graph adhere to UPI (Uniform Protocol Interface). This can be seen clearly in Figure 2.5 [1].

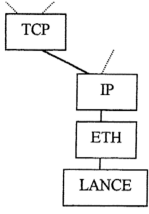

Figure 2.4 A protocol graph [1].

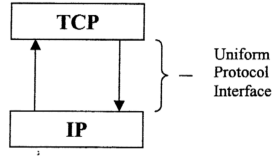

Figure 2.5 The Uniform Protocol Interface [1].

The x-kernel can implement network protocols in a simulation mode, as a user level mode or in a stand-alone mode [5]. The simulation mode and the user level mode allow the user to implement and test the protocols before putting it into the kernel. This gives an

opportunity to test and tweak the network protocols before the real implementation. In the simulation or the user level, compiling of the protocols objects into network modules is much more straightforward, as it is implemented in the user level and not in the operating system kernel itself. Protocols can be built, composed and tested before porting the respective protocols into the operating system kernel. As for the stand-alone mode the x-kernel will be ported into the operating system kernel and it behaves exactly the same as a network implementation in a kernel. The stand-alone mode allows two hosts to communicate with each other without the help of the standard network kernel processes in the operating system.

The x-kernel behaves as a protocol implementation framework that enables one to test, compare and implement network-based protocols relatively easier than the traditional way of network kernel programming. Basically the x-kernel sets a platform to implement protocol modules and protocol stacks [1].