Chapter 3.0 The x-kernel architecture

The x-kernel architecture views a protocol as a specification of a communication abstraction through which collections of participants exchange a set of messages. While the protocol's specification defines what it means to send or receive a message using the protocol's service interface, the x-kernel defines the precise way in which these operations are invoked in a given system with a set of routines and library [5]. Three main communication objects to support the x-kernel model, the objects are protocols, sessions and messages. A set of support routines work in par with the three communication objects; which are protocol, session and message objects to implement protocols [3]. The entire communication system is embedded in the kernel, in effect; the x-kernel's object-oriented infrastructure forms the kernel of the system, with individual protocols configured in as needed. Protocols can be coded and debugged in the simulator or user-level, and then move them, unchanged, to the stand-alone kernel [1]. The x-kernel simulator has the same architecture and implementation environment as for the x-kernel itself.

3.1 The x-kernel architecture explained

The x-kernel architecture core functions lie in the protocol objects, session objects, message objects with the combination of a set of support routines. The relationship between protocols and sessions occur when message objects bind them together.

3.1.1 Protocol objects

Protocol objects serve two main functions. Firstly, it is used to create session objects and secondly to de-multiplex messages received from lower level session objects. This is done on behalf of the former protocol object. A protocol object supports three operations for creating session objects and another operation from the lower level to switch between sessions.

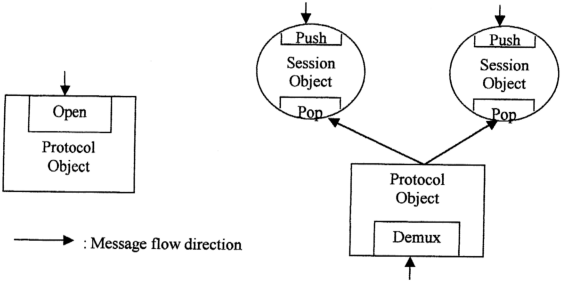

Figure 3.1 Instances of Protocol and Session objects [3].

The three support routines used to create session objects are:

session = open (protocol, invoking_protocol, participant_set)

open_enable (protocol, invoking_protocol, participant_set)

session open_done (protocol, invoking_protocol, participant_set)

There are three main operations involved in communicating between protocols. These three operations are:

(i) A high-level protocol invokes a low-level protocol's *open* operation to create a session. This session is said to be in the low-level protocol's class and created on behalf of the high-level protocol. Each protocol object is given a low-level capability at configuration time.

(ii) The capability for the invoking protocol is passed to the *open* operation. It serves as the newly created session's handle for that protocol.

(iii) In the third operation, the high-level protocol (*open_enable*) passes a capability of itself to a low-level protocol. The latter protocol then invokes the former protocol's *open_done* operation to inform the high-level protocol that it has created a session on its behalf.

From this, it can be seen that the first operation supports session creation triggered by a user process. While the second and third operations, taken together, supports session creation triggered by a message arriving from the network. The *participant_set* argument in all three operations identifies the set of participants that are to communicate via the created session. In the case of *open* and *open_done*, all members of the participant set must be given. Whereas in *open_enable* not all the participants need to be specified. Participants identify themselves and their peers with host addresses, port numbers and protocol numbers. These identifiers are called external identifiers. Each protocol object's *open* and *open_enable* operations use the map routines to save bindings of these external identifiers for capabilities of session objects. Such capabilities for operating system objects are known as internal identifiers.

The support routine to switch between session objects:

demux (protocol, message)

In addition to creating sessions, each protocol also switches messages received from the network to one of its sessions with a *demux* operation. The *demux* takes a message as an argument, and either passes the message to one of its sessions, or creates a new session using the *open_done* operation and then passes the message to it. In the case of a protocol like IP, *demux* might also route the message to some other lower-level session. Each protocol object's *demux* operation makes the decision as to which session should receive the message by first extracting the appropriate external ids from the message's header. It then uses a map routine to translate the external identifiers into either an internal identifier for one of its sessions in which case *demux* passes the message to that session or into an internal id for some high-level protocol in which case *demux* invokes that protocol's *open_done* operation and passes the message to the resulting session.

3.1.2 Session objects

A session is an instance of a protocol created at runtime as a result of an *open* or an *open_done* operation. Intuitively, a session corresponds to the end-point of a network connection; it interprets messages and maintains state information associated with a connection. For example, TCP session objects implement the sliding window algorithm and associated message buffers, IP session objects fragment and reassemble datagrams, and User Datagram Protocol (UDP) session objects only add and strip UDP headers.

Sessions support two primary operations:

push(session, message)

pop(session, message)

The first is invoked by a high-level session to pass a message down to some low-level session. The second is invoked by the *demux* operation of a protocol to pass a message up to one of its sessions.

3.1.3 Message objects

Conceptually, messages are active objects. It either arrives at the bottom of the kernel at a device and flows upward to a user process; or it arrives at the top of the kernel as a user process generates them and flows downwards to a device. While flowing downwards, a message visits a series of sessions via its *push* operations. Whereas, while flowing upwards, a message alternatively visits a protocol via its *demux* operation and then a session in that protocol's class via its *pop* operation. As a message visits a session on its way down, headers are added, the message may fragment into multiple message objects, or the message may suspend itself while waiting for a reply message. As a message visits a session on the way up, headers are stripped, the message may suspend itself while waiting to reassemble into a larger message, or the message may serialize itself with sibling messages. The data portion of a message is manipulated where headers attached or stripped, fragments created or reassembled using the buffer management.

When an incoming message arrives at the network or kernel boundary, the network device interrupts and a kernel process is dispatched to shepherd it through a series of protocol and session objects. This process begins by invoking the lowest-level protocol's *demux* operation. Should the message eventually reach the user/kernel boundary, the shepherd process does an upcall and continues executing as a user process. The kernel process is returned and made available for reuse whenever the initial protocol's *demux* operation returns or the message suspends itself in some session object. In the case of

outgoing messages, the user process does a system call and becomes a kernel process. This process then shepherds the message through the kernel. Thus, when the message does not encounter contention for resources, it is possible to send or receive a message with no process switches. Finally, messages that are suspended within some session object can later be re-activated by a process created as the result of a timer event.

3.1.4 Relationship between protocol and sessions

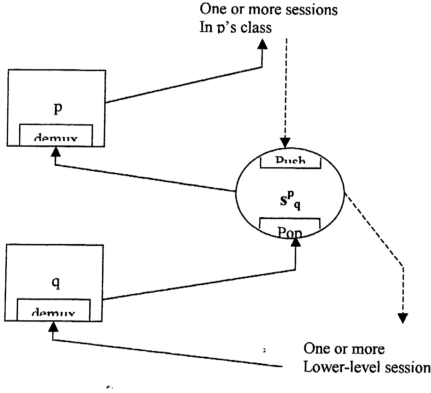

Figure 3.2 Relationship between Protocols and Sessions with Message flow [3].

Consider, the example from figure 3.2. A high-level protocol object, p that depends on a low-level protocol object, q and a session object s^p_q.

(i) When p is invoked by a higher level protocol, it causes p to invoke q's *open* operation with the particular set *{local port, remote port}*.

(ii) The implementation of q's *open* would initialize a new session *s* and save the binding *{local port, remote port}* s in a map.

(iii)Similarly, should p invoke q's *open_enable* operation with a single participant set *{local port}*, q's implementation of *open_enable* would save the binding *{local port}* p in a map.

With, given the invocations of *open* and *open_enable* outlined above, q's *demux* operation would first extract the *local port* and *remote port* fields from the message header and attempt to map the pair *{local port, remote port}* into some session object s. If successful, it would pass the message on to session s. If unsuccessful, q's *demux* would next try to map *local port* into some protocol object p. If the map manager supports such a binding, q's *demux* would then invoke p's *open_done* operation with the participant set *{local port, remote port}* yielding some session s and then pass the message on to s.

As for the session object, figure 3.2 schematically depicts the situation. Session object, s^{P}_{q} that is in protocol q's class was created either directly via *open* or indirectly via *open_enable.* A message can either travel from a user process down to a network device or from a network device up to a user process. The arrows in figure 3.2 show the direction in which the message flows.

3.1.5 The x-kernel support routines

The x-kernel provides a set of support routines that protocol implementations can call to perform the tasks that they must perform, for example, manipulate messages, scheduling events, and map identifiers. The key observation is that protocols look alike in many aspects. The x-kernel simply codifies these common features and makes them

available for protocol implementation in a set of support routines. One advantage of this framework is that it does not need to recode from scratch [2]. For example, a single event manager can be used by all protocols. Furthermore improvement in the algorithm of data structure can be used to support routines that benefit all protocols.

3.1.5.1 The support routines description

(i) Message Library

The message library provides a set of efficient, high-level operations for manipulating messages. The underlying data structure that implements messages is optimized for fragmentation/reassembly, and for adding/stripping headers.

(ii) Participant Library

Participant lists identify members of a session and are used for opening connections. An upper protocol interested in establishing a connection constructs a participant list and passes it to a lower protocol as a parameter of an open routine. The lower protocol then extracts information from the participant list, possibly passing the participant list onto its own lower protocol. Each participant in the list contains a participant address stack; designed to facilitate a general method of communication encapsulated address information between protocol layers. By using pointers to address information, one layer can pass address information through a lower layer without having the lower layer manipulate the address information at all, not even by copying. The address information for each participant is kept as a stack of (*void* *) pointers to address components and the lengths of each component. The component pointers are pushed or popped onto the stack by utility functions.

(iii) Event Library

The event library provides a mechanism for scheduling a procedure to be called after a certain amount time. By registering a procedure with the event library, protocols are able to timeout and act on messages that have not been acknowledged or to perform periodic maintenance functions.

(iv) Map Library

The map library provides a facility for maintaining a set of bindings between identifiers. The map library supports operations for adding new binding to the set, removing bindings from the set, and mapping one identifier into another, relative to a set of bindings. Protocol implementations use these operations to translate identifiers extracted from message headers like the addresses and port numbers into capabilities for pointers to x-kernel objects such as *Protl, Sessn, Enable* objects.

(v) Thread Library

The x-kernel uses a thread-per-message model of computation, and provides primitives for synchronizing threads. The thread function *semWait* is mainly used to increment the use-count for the semaphore. This then allows the x-kernel to run on a different thread. The x-kernel threads are created and destroyed implicitly. Threads are created by the device driver in the case of incoming messages, by the system call in the case of outgoing messages and by the event library in the case of an event firing. Threads are destroyed when they return from the outer-most procedure.

(vi) Trace Library

The trace facility is used for tracing protocol execution. Every protocol should make use of the trace facility. In addition to the trace facilities that print information to standard output, as described in the previous section, the x-kernel also provides a

facility for saving detailed trace information about protocol execution to disk. Various protocol-specific analysis tools can later process this data. This tracing facility supports operations for creating and managing circular trace buffers, writing trace entries to a buffer, saving traces to a file and appending post-amble information to trace files.

(vii) Utility Routines

This utility mainly consists of parsing routines which are used to transfer data from storage to *ROM* files. *ROM* file is configurable file, which is configured to support protocol implementations.

(viii) Control Operations

Control operations are used to perform arbitrary operations on protocols and sessions, via the *xControlProtl* and *xControlSessn* operations described in [2]. These operations return an integer that indicates the length in bytes of the information, which was written into the buffer, or –1 to indicate an error. All implementations of control operations should check the length field before reading or writing the buffer, returning –1 if the buffer is too small. The *checkLen*(*actualLength, expectedLength*) macro can be used for this. The *opcode* field in the control operations specifies the operation to be performed on the protocol or session. There are two types of operations: standard ones that may be implemented by more than one protocol and protocol specific ones [2].

This discussion on the support routines by the x-kernel only gives a brief overview of what is available, for more detailed information of these routines is found in [2]. The real

implementation of the x-kernel objects discussed here can be seen in *ip.c* program listing in *Appendix C*.

3.2 The implementation environment

After coding the appropriate programs for protocols, compiling it and creating the objects files ready for implementation is the next step in implementation process. The x-kernel can be configured into three main implementation environments; which are in the user-level, stand-alone and simulator mode [5]. For each and every implementation environment, the x-kernel basically goes through a similar process of configuration.

3.2.1 Network Protocols at User Level

There are several factors that motivate protocol implementations that are outside kernel. The most obvious of these are ease of prototyping, debugging and maintenance. Two more factors are:

1. The co-existence of multiple protocols that provide materially differing services, and the clear advantages of easy addition and extensibility by separating their implementations into self-contained units.
2. The ability to exploit application-specific knowledge for improving the performance of a particular communication protocol.

3.2.1.1 Multiplicity of Protocols

Over the years, there has been a proliferation of protocols driven primarily by application needs. For example, the need for an efficient transport for distributed systems was a factor in the development of request/response protocols in lieu of existing byte-stream protocols such as TCP. However these protocols do not always deliver the highest throughput. In systems that need to support both throughput-intensive and latency-critical applications, it is realistic to expect both types of protocols to co-exist.

Future uses of workstation clusters as message passing multi computers will undoubtedly influence protocol design: efficient implementations of this and other programming paradigms will drive the development of new transport protocols. As newer networks with different speed and error characteristics are deployed, protocol requirements will change. Different network links exist at a single site, multiple protocols may need to co-exist.

3.2.1.2 Exploiting Application Knowledge

In addition to using special purpose protocols for different application areas, further performance advantages may be gained by exploiting application-specific knowledge to fine tune a particular instance of a protocol. Watson and Mamrak have observed that conflicts between application-level and transport-level abstractions lead to performance compromises [16]. One solution to this is to describe a general-purpose protocol with respect to a particular application. In this approach, based on application requirements, a specialized variant of a standard protocol is used rather than the standard protocol itself. A different application would use a slightly different variant of the same protocol. Language-based protocol implementations as well as protocol compilers are two recent attempts at

exploiting user specified constraints to generate efficient implementations of communication protocols. In particular, the notion of specializing a transport protocol to the needs of a particular application has been the motivation behind many recent system designs.

3.2.1.3 Alternative Protocol Structures

The discussion above argues for alternatives to monolithic protocol implementations since they are deficient in at least two ways. First, having all protocol variants executing in a single address space (especially if it is in-kernel) complicates code maintenance, debugging, and development. Second, monolithic solutions limit the ability of a user (or a mechanized program) to perform application-specific optimizations. In contrast, given the appropriate mechanisms in the kernel, it is feasible to support high performance and secure implementations of relatively complex communication protocols as user-level libraries. Figure 3.3 and 3.4 shows different alternatives for structuring communication protocols.

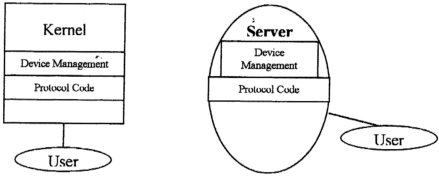

Figure 3.3 Monolithic Organizations

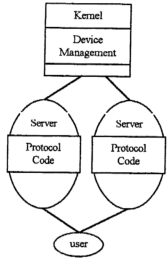

Figure 3.4 Non-monolithic Organizations

Traditional operating systems like UNIX and modern micro kernels such as Mach 3.0 have similar monolithic protocol organizations. The Mach 3.0 micro kernel implements protocols outside the kernel within a trusted user-level server. The code for all system-supported protocols runs in the single, trusted, UX server's address space. There are at least three variations to this basic organization depending on the location of the network device management code and the way in which the data is moved between the device and the protocol server. In one variant of the system, the Mach/UX server maps network devices into its address space, has direct access to them and is functionally similar to a monolithic in-kernel implementation. In the second variant, device management is located in the kernel. The in-kernel device driver and the UX server communicate through a message based interface. One alternative to a monolithic implementation is to dedicate a separate user-level server for each protocol stack and separate server(s) for network device management. This arrangement has the potential for performance problems since the critical send/receive path for an application could incur excessive domain-switching overheads because of address space crossings between the user, the protocol server and the

device manager. That is, given identical implementations of the protocol stack and support functions like buffering, layering and synchronization, inter-domain crossings come at a price. Further, and perhaps more importantly, this arrangement, like the monolithic version, does not permit easy exploitation of application-level information.

This system implements packet demultiplexing and device management within the kernel. Besides this, it also supports implementations of standard protocols such as TCP and VMTP outside the kernel. It does not rely on any special-purpose hardware or on extensive operating system support.

In the common case of sends and receives, the library talks to the device manager without involving a dedicated protocol server as an intermediary. Implementation application level protocols beneath a UNIX compatible interface shares many features of implementation which enforces less control on outgoing packets and does not provide the full semantics of the UNIX socket interface, meaning that not all existing UNIX programs will work with this implementation. It would be easy to combine the two implementations into one that has neither of these deficiencies. In general, there are several alternatives to distributing the implementation of a set of protocols among a set of address spaces (e.g., the application, a trusted server, the kernel). Each resulting organization leads to tradeoffs in performance, protection, ease of debugging, etc. This paper describes the design and implementation of one such organization where the protocol suite is located in a user level library and compares it with the in-kernel and single server alternatives. Current research at the University of Arizona tries to address the general question of protocol decomposition into multiple domains in the context of the x-kernel [3]. Portioning an x-kernel protocol graph among different address spaces allows performance and trust tradeoffs of various protocol organizations to be easily explored.

3.2.2 User Level Implementations

The Build directory for the user-level defines three main files for construction of the x-kernel; which are (i) *graph.comp*, (ii) *prottab* and (iii) *rom* files [2].

(i) Specifying a Protocol Graph: *graph.comp* specifies the collection of protocols that are to be included in the kernel and the relations between them. This file is divided into three sections; device drivers, protocols and miscellaneous configuration parameters.

(ii) Protocol Tables: *prottab* a protocol table file that defines the number space for protocols to identify each other, it identifies the relative port numbers. The x-kernel builds a table of protocol numbers by reading configuration files at runtime and provides operations for protocols to query this table to determine the protocol corresponding to the protocol number. Thus, it is not necessary to embed explicit protocol numbers in the protocol code itself.

(iii) *ROM* files: *.rom* files specify run time options, such as IP address for protocols of the machine on which an x-kernel will be run. This file allows specification or runtime options for protocols and various subsystems. Each rom entry consists of a single line. The first field in each line specifies the particular protocol of a subsystem that should interpret that line. The rest of the fields are specific to that particular protocol or subsystem.

3.2.3 Simulator Implementation

The network simulator based on the x-kernel provides a framework for developing, analyzing and testing network protocols. The x-kernel provides the tool x-sim to test network protocols. The x-sim is tightly integrated with the x-kernel architecture; it runs x-

kernel protocols and protocols can be moved between the simulator and the x-kernel without changes. In the simulator a special protocol called *SIM* layer is included at the bottom of all the host protocol stacks. This will be explained later in chapter 5 with the IPv6 protocol for simulation issues.

The *graph.comp* file is still used in the simulator, but only to specify which protocol modules to include in the executable. At runtime the simulator reads a configuration file called *xsim.data*, which specifies the network topology.

In summary, building the protocol graph for a simulation involves two stages: including the necessary protocol modules in the simulator executable (via *graph.comp*) and specifying at runtime the protocol graphs for the individual simulated hosts (via *xsim.data*). [6].

(i) Internet Configuration: *xsim.data*

This file specifies most aspects of a simulation, including the structure or the simulated internetwork and the tests to be run. This file contains a set of entries of each, which describes an Internet component such as network, routers, or hosts and defines macro variables.

(ii) ***graph.comp***

This file serves the same purpose as described for the user-level implementation, but in this file the relationship between protocols specified are ignored by the simulator. This file is only as a means of specifying which protocols module to include during runtime.

Finally, after configuring the configuration files in both in the simulation and user-level these files go through the compilation process. The compilation process is followed

by an installation and configuring of the x-kernel and x-sim into Linux process. The steps in building the executables is almost similar for both cases in its particular directories, except for some small tweaks in the make command procedure of the compiling process. The user-level and simulator implementation gives an opportunity for the protocol developer to test and run simulations before implementing the protocols in a stand-alone kernel. The simulator can also be used to just run new protocols from a basic internetwork structure using a host to host network using Point-to-Point link or an Ethernet to investigate IP implementations. In this context the issues related with IPv6 is based on the minimal specification of RFC2460.

3.3 Designs and Implementation of User-Level Protocols

3.3.1 Design Overview

This section describes the design phase at a higher level. In this design, protocol functionality is provided to an application by three interacting components.

(i) A protocol library that is linked into the application.

(ii) A registry server that runs as a privileged process.

(iii) A network I/O module that is co-located with the network device driver.

Figure 3.5 shows an overall view of our design and the interaction between the components.

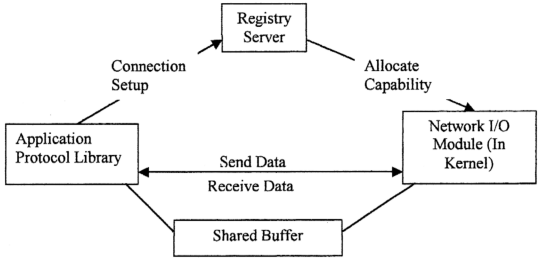

Figure 3.5 Structure of the Protocol Implementation

The library contains the code that implements the communication protocol. For instance, typical protocol functions such as retransmission, flow control, check summing, are located in the library. Given the timeout and retransmission mechanisms of reliable transport protocols, the library typically would be multithreaded. Applications may link to more than one protocol library at a time. For example, an application using TCP will typically link to the TCP, IP, and ARP libraries.

The registry server handles the details of allocating and deallocating communication end-points on behalf of the applications. Before applications can communicate with each other, they have to be named in a mutually secure and non-conflicting manner. The registry server is a trusted piece of software that runs as a privileged process and performs many of the functions that are usually implemented within the kernel in standard protocol implementations. There is a dedicated registry server for each protocol.

The third module implements network access by providing efficient and secure input packet delivery, and outbound packet transmission. There is one network I/O module

for each host-network interface on the host. Depending on the support provided by the host-network interface, some of the functionality of this module may be in hardware. Given the library, the server, and the network I/O module, applications can communicate over the network in a straightforward fashion. Applications call into the library using a suitable interface to the transport protocol (e.g., the BSD socket). The library contacts the registry server to negotiate names for the communication entities. In connection-oriented protocols this might require the server to complete a connection establishment protocol with a remote entity. Before returning to the library, the registry server contacts the network I/O module on behalf of the application to set up secure and efficient packet delivery and transmission channels. The server then returns to the application library with un-forgeable tickets or capabilities for these channels. Subsequent network communication is handled completely by the user-level library and the network I/O module using the capabilities that the server returned. Thus, the server is bypassed in the common path of data transmission and reception.