x-kernel Tutorial

Larry L. Peterson
Bruce S. Davie
Andrew C. Bavier

January 1996

Abstract

This document is a tutorial on writing *x*-kernel protocols. It assumes a basic understanding of network protocols. This document does not cover the entire *x*-kernel interface, focusing instead on the *x*-kernel's most common operations. For a complete description of the *x*-kernel, see [3]. This document also does not describe how to configure and run the *x*-kernel, focusing instead on how to write the individual protocols that one might want to configure into a given instance of the *x*-kernel. For an introduction on how to configure and run the *x*-kernel, see [4]. Finally, this document borrows heavily from [5], which gives both an overview of the *x*-kernel and a general introduction to computer networks.

Contents

1 Object-Based Protocol Implementation

The x-kernel provides an object-based framework for implementing protocols, that is, the key abstractions of the x-kernel are represented by objects. An object may be conveniently thought of as a data structure with a collection of operations that are *exported*, that is, made available for invocation by other objects. Objects that have similar features are grouped into classes, with two obvious object classes for a protocol implementation environment being the *protocol* and the *message*.

To better understand the role played by the x-kernel, think of a protocol as an abstract object, one that exports both a *service interface* and a *peer-to-peer interface*. The former defines the operations by which other protocols on the same machine invoke the services of this protocol, while the latter defines the form and meaning of messages exchanged between peers (instances of the same protocol running on different machines) to implement this service. In a nutshell, the x-kernel provides a concrete representation for a protocol's service interface. 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. For example, the x-kernel's operations for sending and receiving a message are xPush and xPop, respectively. An x-kernel protocol object would consist of an implementation of xPush and xPop that adheres to what the protocol specification says it means to send and receive messages using this protocol. In other words, the protocol specification defines the *semantics* of the service interface, while the x-kernel defines one possible *syntax* for that interface.

The fashionableness of object-based programming has been accompanied by a proliferation of object-oriented languages, of which C++ is probably the most well known example. The x-kernel, however, is written in C, which is not considered to be an object-oriented language. To deal with this, the x-kernel provides its own object infrastructure—the glue that makes it possible for one object to invoke an operation on another object. For example, when an object invokes the operation xOpen on some protocol p (xOpen is the x-kernel operation for opening a communication channel) it includes p as one of the arguments to this operation. The x-kernel's object infrastructure takes care of invoking the actual procedure that implements this operation on protocol p. In other words, the x-kernel uses xOpen(p, ...) in place of the more conventional notation for invoking an operation on an object: p.xOpen(...).

2 Protocols and Sessions

The two main classes of objects supported by the x-kernel are *protocols* and *sessions*. Protocol objects represent just what you would expect—protocols such as IP or TCP. Session objects represent the local end-point of a channel, and as such, typically implement the code that interprets messages and maintains any state associated with the channel. The protocol objects available in a particular network subsystem, along with the relationships among these protocols, is defined by a protocol graph at the time a kernel is configured. Session objects are dynamically created as channels are opened and closed. Loosely speaking, protocol objects export operations for opening channels—resulting in the creation of a session object—and session objects export operations for sending and receiving messages.

The set of operations exported by protocol and session objects is called the *uniform protocol interface*—it defines how protocols and sessions invoke operations on each other. At this stage, the important thing to know about the uniform protocol interface is that it specifies how high-level objects invoke operations on low-level objects to send outgoing messages, as well as how low-level objects invoke operations on high-level objects to handle incoming messages. For example, consider the specific pair of protocols TCP and IP in the Internet architecture. TCP sits directly above IP in this architecture, so the x-kernel's uniform protocol interface defines the operations that TCP invokes on IP, as well as the operations IP invokes on TCP, as illustrated in Figure 1.

Keep in mind that the following discussion defines a common *interface* between protocols, that is, the operations one protocol is allowed to invoke on the other. This is only half the story, however. The other half is that each protocol has to provide a routine that *implements* this interface. Thus, for an operation like xOpen, protocols like TCP and IP must include a routine that implements xOpen; by convention, we name this routine tcpOpen and ipOpen, respectively. Therefore, the following discussion not only defines the interface to each operation, but it also gives a rough outline of what every protocol's implementation of that operation is expected to do.

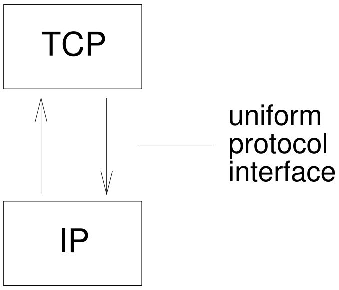


Figure 1: Uniform Protocol Interface.

2.1 Configuring a Protocol Graph

Before presenting the operations that protocol and session objects export, we first explain how a protocol programmer configures a protocol graph. Standardization bodies like the ISO and the IETF define a particular network architecture that includes a specific set of protocols. In the Internet architecture, for example, TCP depends on IP, by definition. This suggests that it is possible to "hard-code" TCP's dependency on IP into the TCP implementation. While this could be done in the case of TCP, the x-kernel supports a more flexible mechanism for configuring a protocol graph. This makes it easy to plug protocols together in different ways. While this is a quite powerful thing to be able to do, one has to be careful that it makes sense to have any two protocols adjacent to each other in the protocol graph.

Quite simply, a user that wants to configure a protocol graph specifies the graph with a text file, called graph.comp, of the following form:

```
name=lance;
name=eth protocols=lance;
name=arp protocols=eth;
name=ip protocols=eth,arp;
name=icmp protocols=ip;
name=udp protocols=ip;
name=tcp protocols=ip;
```

This specification results in the protocol graph depicted in Figure 2. In this example graph, lance and eth combine to implement an Ethernet device driver: lance is the device-specific half and eth is the device-independent half. Also, arp is the Address Resolution Protocol (it is used to translate IP addresses into Ethernet addresses) and icmp is the Internet Control Message Protocol (it sends error messages on behalf of TCP and IP). The name field in each line specifies a protocol (by name) and the protocols field says which other protocols this protocol depends on. Not shown in this example is a dir field that identifies the directory where the named protocol implementation can be found; by default, it is the same as the name of the protocol. The x-kernel build program, called compose, parses this specification, and generates some C code that initializes the protocol graph when the system is booted.

2.2 Operations on Protocol Objects

The primary operation exported by a protocol object allows a higher level entity to *open* a channel to its peer. The return value from an open operation is a session object. Details about the session object are discussed in the following subsection. For now, think of a session as a convenient object for gaining access to the channel; the module that opened the session object can send and receive messages using the session object. An object that lets us gain access to something abstract is sometimes called a *handle*—you can think of it as the thing that makes it easy to grab something that is otherwise quite slippery. Thus, a session object provides a handle on a channel.

In the following discussion, we need a generic way to refer to the entity that opens a channel, since sometimes it is an application program, and sometimes its another protocol. We use the term *participant* for this purpose. That is,

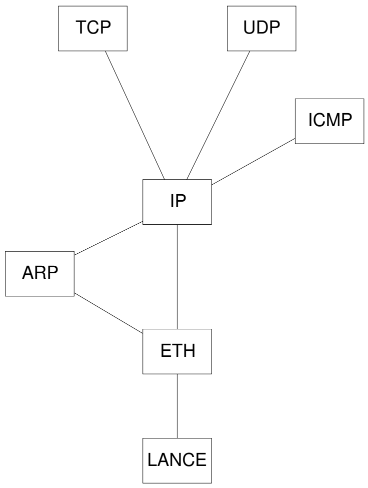


Figure 2: Example Protocol Graph.

we think in terms of a pair of participants at level i communicating over a channel implemented by a protocol at level $i-1$ (where the level number decreases as you move down the stack).

Opening a channel is an asymmetric activity. Generally, one participant initiates the channel (we call it the client). This local participant is therefore able to identify the remote participant, and is said to do an *active* open. In contrast, the other participant accepts the channel (we call it the server). This participant does not know what clients might try to talk to it until one of them actually makes contact. The server, therefore, does a *passive* open—it says to the lower level protocol that it is willing to accept a channel, but does not say with whom the channel will be.

Thus, the exact form of the open operation depends on whether the higher level entity is doing an active open or a passive open. In the case of an active open, the operation is:

```
Sessn xOpen(Protl hlp, Protl hlpType, Protl llp, Part *participants)
```

This operation says that high-level protocol hlp is opening low-level protocol llp so as to establish a channel with the specified participants. For a typical channel between a pair of participants, this last argument would contain both the local participant's address and the remote participant's address. The low-level protocol does whatever is necessary to establish the channel, which quite often implies opening a channel on a still lower-level protocol. Notice that Protl and Sessn are the C type definitions for protocol and session objects, respectively.

The hlpType argument to xOpen is a bit subtle. What is really happening is that hlp is opening a session associated with llp *on behalf of* high-level protocol hlpType. Typically, hlp and hlpType refer to the same protocol, although as we will see in Section 9.5, there are some cases in which hlp and hlpType are not equivalent.

A high-level protocol passively opens a low-level protocol with a pair of operations:

```
XkReturn xOpenEnable(Protl hlp, Protl hlpType, Protl llp, Part *participant)

XkReturn xOpenDone(Protl hlp, Protl llp, Sessn session, Part *participants)
```

xOpenEnable is used by high-level protocol hlp to inform low-level protocol llp that it is willing to accept a connection. (Argument hlpType has the same meaning as in xOpen.) In this case, the high-level protocol usually specifies only a single participant—itself. The xOpenEnable operation returns immediately; it does not block waiting for a remote site to try to connect to it. The low-level protocol remembers this enabling; when some remote participant subsequently connects to the low-level protocol, llp calls the high-level protocol's xOpenDone operation to inform it of this event. The low-level protocol llp passes the newly created session as an argument to high-level protocol hlp, along with the complete set of participants, thereby informing the high-level protocol of the address for the remote entity that just connected to it. XkReturn is the return value of all the uniform protocol interface operations presented here except for xOpen and xPush; it indicates whether the operation was successful (XK_SUCCESS) or not (XK_FAILURE).

In addition to these operations for opening a connection, x-kernel protocol objects also support an operation for demultiplexing incoming messages to the appropriate channel (session). In this case, a low-level session invokes this operation on the high-level protocol that at some earlier time had opened it. The operation is:

```
XkReturn xDemux(Protl hlp, Sessn lls, Msg *message)
```

It will be easier to understand how this operation is used after we look at session objects in more detail.

2.3 Operations on Session Objects

As already explained, a session can be thought of as a handle on a channel that is implemented by some protocol. One can also view it as an object that exports a pair of operations: one for sending messages, and one for receiving messages:

```
XkHandle xPush(Sessn lls, Msg *message)

XkReturn xPop(Sessn hls, Sessn lls, Msg *message, void *hdr)
```

The implementation of xPush and xPop is where the real work of a protocol is carried out—it's where headers are added to and stripped from messages, and then interpreted. In short, these two routines implement the algorithm that defines the protocol.

The operation of xPush is fairly straightforward. It is invoked by a high-level session to pass a message down to some low-level session (lls) that it had opened at some earlier time. lls then goes off and does what is needed with the message—perhaps using xPush to pass it down to a still lower level session. This is illustrated in Figure 3. In this figure we see three sessions, each of which implements one protocol in a stack, passing a message down the stack using xPush.

Passing messages back up the stack using xPop is more complicated. The main problem is that a session does not know what session is above it—all it knows is the protocol that is above it. So, a low-level session lls invokes the xDemux routine of the protocol above it. That protocol, since it did the work of opening the high level session hls to which this message needs to go, is able to pass the message to hls using its xPop routine. How does a protocol's xDemux routine know which of its potentially many sessions to pass the message up to? It uses the *demultiplexing key* found in its header.

In addition to the hls that is being called and the message being passed to it, xPop takes two other arguments. First, lls identifies the low-level session that handed up this message via xDemux. Second, since xDemux has to inspect the message header to select the session on which to call xPop—i.e., it has already gone to the effort of extracting the protocol's header—it passes the header (hdr) as the final argument to xPop. This chain of events is illustrated in Figure 4.

To see how this works in practice, imagine we want to send a message using the TCP and IP protocols. An application program opens a channel by performing xOpen on TCP; TCP returns a session object to the application. TCP opens an IP channel by performing xOpen on IP; IP returns a session object to TCP. When the application wants to send a message, it invokes the xPush operation of the TCP session; this session in turn invokes the xPush operation of the IP session, which ultimately causes the message to be sent.

Now suppose an incoming message is delivered to the IP session. This session has no idea about the TCP session above it, so it does the only thing it knows how to do—it passes the message up to the TCP *protocol* using xDemux. The TCP protocol knows about all the TCP sessions, and so passes the message up to the appropriate TCP session using xPop. TCP's xDemux uses the demux key it found in the TCP header to select among all the TCP sessions.

The final operation that we need to be able to perform is one to close a session, which in effect closes the channel to the other machine.

```
XkReturn xClose(Sessn session)
```

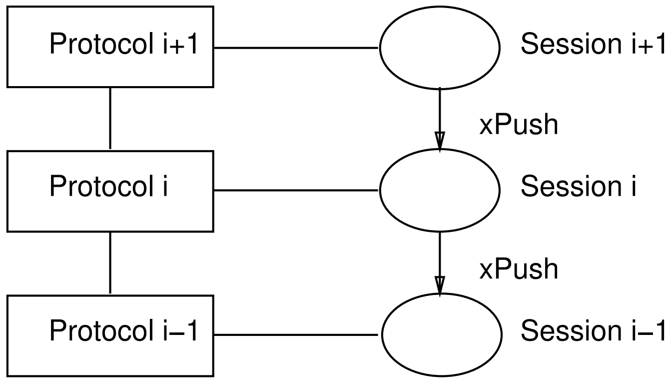


Figure 3: Using xPush to pass a message down a stack.

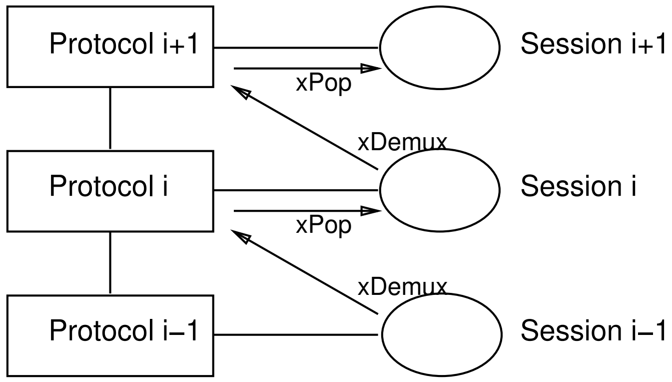


Figure 4: Using xDemux and xPop to pass a message up a stack.

In addition to sessions and protocols, this discussion has introduced two other x-kernel object classes: messages and participants. Both classes represent exactly what you think they do—the messages that protocols send to each other (corresponding to type definition Msg), and the addresses of participants that are communicating over some channel (corresponding to type definition Part). Message and participant objects are discussed in more detail in later sections.

2.4 Asynchronous versus Synchronous Protocols

As described so far, the x-kernel supports *asynchronous* protocols—protocols that do not block waiting for a reply from their peer. Some protocols, however, are *synchronous*—the caller blocks until a reply can be returned. Clearly, the xPush/xDemux/xPop paradigm just described is not going to work for synchronous protocols since it makes no provision for a return value. The x-kernel accommodates synchronous protocols by providing a parallel set of operations for sending and receiving messages:

```
XkReturn xCall(Sessn session, Msg *request, Msg *reply)

XkReturn xCallPop(Sessn session, Msg *request, Msg *reply, void *hdr)

XkReturn xCallDemux(Protl hlp, Sessn session, Msg *request, Msg *reply)
```

The key difference, of course, is that each operation now returns a reply message (reply); for clarity, we refer to the message given as an argument as request. The operations are synchronous in the sense that each cannot return until the reply message is available.

So far so good: some protocols are purely asynchronous (they export xPush, xPop and xDemux operations), and some are purely synchronous (they export xCall, xCallPop and xCallDemux operations). However, if all protocols were either asynchronous or synchronous, then the entire protocol graph would have to consist of only asynchronous or synchronous protocols—an asynchronous protocol can only call xPush on an adjacent protocol, meaning it could never be composed with a synchronous protocol.

Fortunately, there can be hybrid protocols that are half synchronous and half asynchronous. This does not mean that they support all six operations, but rather they look like a synchronous protocol to higher level protocols, and like an asynchronous protocol to lower level protocols. Such a protocol supports the xCall operation rather than xPush on top, while from below, it still supports the asynchronous xDemux/xPop interface, that is, it turns an underlying asynchronous communication service into a synchronous communication service. It does this by having the sending process (caller) block on a semaphore waiting for a reply message.

2.5 Process Models for Protocols

As we have said, protocol implementors typically have to be concerned about a lot of operating system issues. This subsection introduces one of the most important of these issues—the process model.

Most operating systems provide an abstraction called a *process*, or alternatively, a *thread*. Each process runs largely independently of other processes, and the OS is responsible for making sure that resources, such as address space and CPU cycles, are allocated to all the current processes. The process abstraction makes it fairly straightforward to have a lot of things executing concurrently on one machine; for example, each user application might execute in its own process, and various things inside the OS might execute as other processes. When the OS stops one process from executing on the CPU and starts up another one, we call this a *context switch*.

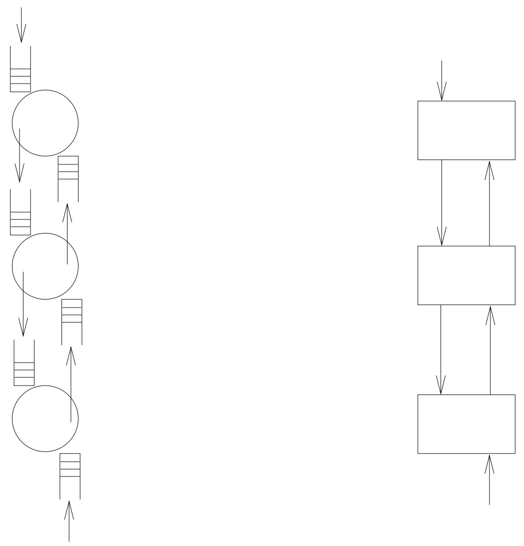
When designing a protocol implementation framework, one of the first questions to answer is: "Where are the processes?" There are essentially two choices, as illustrated in Figure 5. In the first, which we call the *process-per-protocol* model, each protocol is implemented by a separate process. This implies that as a message moves up or down the protocol stack, it is passed from one process/protocol to another—the process that implements protocol i processes the message, then passes it to protocol $i-1$, and so on. How one process/protocol passes a message to the next process/protocol depends on the support the host OS provides for interprocess communication. Typically, there is a simple mechanism for enqueuing a message with a process. The important point, however, is that a context switch is required at each level of the protocol graph—typically a time-consuming operation.

The alternative, which we call the *process-per-message* model, treats each protocol as a static piece of code, and associates the processes with the messages. That is, when a message arrives from the network, the OS dispatches a process to be responsible for the message as it moves up the protocol graph. At each level, the procedure that implements that protocol is invoked, which eventually results in the procedure for the next protocol being invoked, and so on. For out-bound messages, the application's process invokes the necessary procedure calls until the message is delivered. In both directions, the protocol graph is traversed in a sequence of procedure calls.

Although the process-per-protocol model is sometimes easier to think about—I implement my protocol in my process and you implement your protocol in your process—the process-per-message model is generally more efficient. This is for a simple reason: a procedure call is an order of magnitude more efficient than a context switch on most computers. The former model requires the expense of a context switch at each level, while the latter model costs only a procedure call per level.

The x-kernel uses the process-per-message model. Tying this model back to the operations outlined above, this means that once a session (channel) is open at each level, a message can be sent down the protocol stack by a sequence of calls to xPush, and up the protocol stack by alternating calls to xDemux and xPop. This asymmetry—xPush going down and xDemux/xPop going up—is unappealing, but necessary. This is because when sending a message out, each layer knows which low-level session to invoke xPush on because there is only one choice, while in the incoming case, the xDemux routine at each level has to first demultiplex the message to decide which session's xPop to call.

Notice that the high-level protocol does not reach down and *receive* a message from the low-level protocol. Instead, the low-level protocol does an *upcall*—a procedure call up the stack—to deliver the message to the high-level protocol. This is because a receive-style operation would imply that the high-level protocol is executing in a process



(a) process−per−protocol (b) process−per−message

Figure 5: Alternative Process Models.

that is waiting for new messages to arrive, which would then result in a costly context switch between the low-level and high-level protocols. By having the low-level protocol deliver the message to the high-level protocol, incoming messages can be processed by a sequence of procedure calls, just as outgoing messages are.

We conclude this discussion of processes by introducing three operations that the x-kernel provides for process synchronization:

```
void semInit(Semaphore *s, int count)

void semSignal(Semaphore *s)

void semWait(Semaphore *s)
```

These operations implement conventional counting semaphores. Specifically, every invocation of semSignal increments semaphore s by one, and every invocation of semWait decrements s by one, with the calling process blocked (suspended) if decrementing s cause its value to become less than zero. A process that is blocked during its call to semWait will be allowed to resume as soon as enough semSignal operations have been performed to raise the value of s above zero. Operation semInit initializes the value of s to count.

3 Message Library

We now turn our attention from how protocols invoke operations on each other, and consider what goes on inside a particular protocol. One of the most common things that a protocol does is manipulate messages. For example, they add headers to, and strip headers from, messages. Another common way in which protocols manipulate messages is to break a single message into multiple fragments, and later to join these multiple fragments back into a single message. This is necessary because most network links allow messages of only a certain size to be transmitted. Thus, a protocol that uses such a link to transmit a large message must first *fragment* the message on the source node, and then *reassemble* the fragments back into the original message on the destination node. We will see examples of protocols that fragment and reassemble messages in later sections.

Because manipulating messages is a basic part of all protocols, the x-kernel defines an abstract data type—called *message* and given by the C type definition Msg—that includes an interface for performing these common operations.

This section presents the x-kernel's message abstraction. (See [2] for a detailed description of how the message library is implemented.)

The message abstraction can best be viewed as a byte string of some length. For the purpose of this discussion, we use the term "message" to refer to the abstract object and we use the term "data" to refer to the actual byte string contained in a message. For example, message m schematically depicted in Figure 6 contains the data "abcdefg".

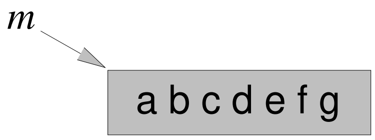


Figure 6: Message object containing a byte string.

In effect, the operations on the message object can be viewed as string manipulations. For example, while processing an outgoing message, each of several protocols may add a header to the message (i.e., two strings are concatenated) and fragment the message into two or more packets (i.e., a string is divided into two substrings). Similarly, while processing an incoming message, each of several protocols may strip headers from the message (i.e., a string is removed from the front of another string) and reassemble message fragments (i.e., two strings are concatenated). In addition, each of several protocols may save references to portions of a message for future use, e.g., to retransmit in the event of an error in the network. Thus, any given byte may be attached to several different strings, removed from several different strings, and referenced by several different protocols.

3.1 Adding and Stripping Headers

As outgoing messages move down the protocol graph, each protocol attaches (pushes) its header onto the front of the message. Similarly, as an incoming message moves up the protocol graph, each protocol strips (pops) its header from the front of the message. The message object supports the following two operations for pushing and popping headers:

```
char *msgPush(Msg *message, int length)

char *msgPop(Msg *message, int length)
```

Both operations return a pointer to a buffer that contains the header. In the case of msgPush, room for length bytes is attached to the front of the message, and a pointer to this memory location is returned. The protocol can then write the header to this location to effectively add the header to the message. In the case of msgPop, length bytes are removed from the front of the message. The protocol can then read the header available at the returned memory location. Figures 7 and 8 illustrate the semantics of the two operations.

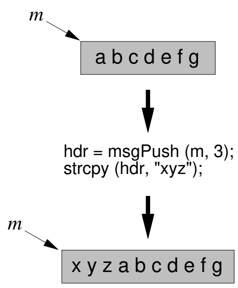


Figure 7: Effects of msgPush operation.

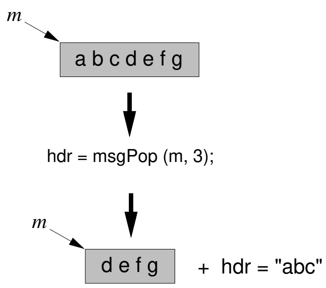


Figure 8: Effects of msgPop operation.

3.2 Fragmenting and Reassembling Messages

Fragmenting and reassembling messages is a common activity in network protocols. The x-kernel supports the following two operations for manipulating message fragments:

void msgBreak(Msg *original_message, Msg *fragment_message, int length)

void msgJoin(Msg *new_message, Msg *fragment1, Msg *fragment2)

The first operation creates a pair of messages by breaking length bytes off the front of the original_message and placing them in fragment_message. After the operation returns, original_message contains the sequence of bytes that remain after length bytes are removed. The second operation attaches fragment1 to the front of fragment2, producing new_message. The arguments to msgJoin need not refer to distinct messages. One common use of msgJoin is to attach a fragment to the end of a larger message, in which case the first two arguments are the same (the larger message) and the third argument is the fragment. These two operations are illustrated in Figures 9 and 10.

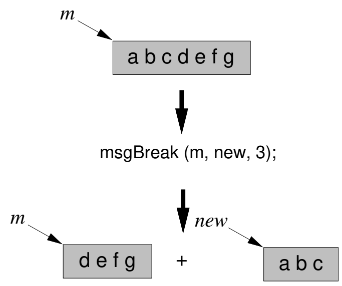


Figure 9: Effects of msgBreak operation.

3.3 Traversing Messages

So as to avoid the unnecessary copying of data from one buffer to another, the message object is implemented by a tree of buffers. (See [2] for a description of this data structure.) Because the data contained in a message object is scattered over multiple, non-contiguous memory buffers, the x-kernel provides a set of operations for walking the tree and extracting the actual data.

void msgWalkInit(MsgWalk cxt, Msg *message)

char *msgWalkNext(MsgWalk cxt, int *len)

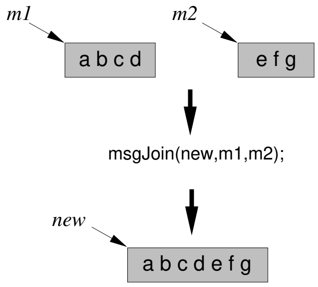


Figure 10: Effects of msgJoin operation.

void msgWalkDone(MsgWalk cxt)

Operation msgWalkNext traverses the message tree, and returns a pointer to the next chunk of data in the message; it also sets len to the number of bytes in that chunk. Argument cxt maintains the context for the message traversal, so that msgWalkNext knows how far through the tree it got on the last invocation. The other two operations—msgWalkInit and msgWalkDone—initialize and destroy this context, respectively.

As a simple example of how one might use msgWalkNext, device drivers often create an array of pointers to the various pieces of the message (along with each piece's length). Arrays of buffer/length pairs are commonly accepted by network devices, so this might be something that is done by a network device driver to prepare an x-kernel message for transmission over a physical link.

3.4 Other Operations

There are additional operations that can be applied to message objects, as summarized below:

void msgConstructEmpty(Msg *message)

void msgConstructBuffer(Msg *message, char *buffer, int length)

char *msgConstructAllocate(Msg *message, int length)

void msgAssign(Msg *message_1, Msg *message_2)

int msgLength(Msg *message)

The first three operations are used to create messages. Each is used under a different set of circumstances. msgConstuctEmpty creates an empty message. It is used in conjunction with msgAssign to save a reference to a message. For example, if a given protocol wants to send a particular message m out over the network, but at the same time save a copy of m in case it needs to retransmit it in the future, it might use msgConstructEmpty to create an empty message n, and then do msgAssign(m, n). At this point both m and n represent the same message (the same byte string).

The other two message constructors create messages with an associated data component. msgConstructBuffer builds a message from the existing byte string referenced by buffer. This operation is used, for example, by an application program that already possesses a buffer of data it wants to transmit; it uses msgConstructBuffer to encapsulate this buffer in a message object. In contrast, msgConstructAllocate is used when a protocol knows it is going to need a message to hold length bytes of data, but it does not yet have the data to place in the message. This operation is used, for example, in a device driver that knows it will eventually receive a packet from the network of some size. It invokes msgConstructAllocate to create the message, and gets a pointer to a memory buffer that is free to hold data in return. The device driver would then program the network adaptor to receive the next packet into this buffer.

4 Participant Library

When a high-level protocol (or an application) opens a channel via some low-level protocol, it identifies the peer with which it wants to communicate by giving the peer's address. More precisely, as seen in Section 1, the high-level protocol passes a participant list—an argument of type Part—to xOpen. The participant list is simply an array of participant addresses where, by convention, the high-level protocol identifies itself in the first element of the array. Although in general the high-level protocol can list an arbitrary number of participants—a group of peers—that are to communicate via the newly established channel, in a unicast (one-to-one) communication, only a single other participant is identified.

4.1 Participant Addresses

Each element of a participant array is actually not just an address, but a *stack* of addresses; each address in the stack is a byte string of some length. The motivation for using a stack of addresses to identify each participant will become clear in a moment. For now, here are the four operations for manipulating participant addresses provided by the participant library:

```
void partInit(Part *participants, int number)

int partLength(Part *participants)

void partPush(Part *participant, char *address, int length)

char *partPop(Part *participant)
```

The first operation initializes a participant list that is to hold the address stacks for the specified number of participants. For a typical one-to-one channel, number is two. The second operation returns the number of participants in the participant list; e.g., two.

The third and fourth operations are used to add (remove) addresses to (from) the address stack of a single participant. partPush pushes the specified address of length bytes onto the stack, while partPop pops the next address off the stack. Giving a length of 0 to partPush denotes a special *wildcard* address, indicating to the low-level protocol that it is free to substitute any meaningful value. This could be used, for example, to allow a server to accept connections from any client. partPop does not include a length field since the returned address is a NULL-terminated character string.

Now to the question of why each participant is identified by a stack of addresses. The short answer is that addresses are too complicated to be stored as a simple byte string. To see why, consider the following. First, each protocol expects participants to identify themselves and their peers with a compound address of the form $\langle$demux key, host address$\rangle$. Second, each protocol defines its own notion of what participant addresses should look like. That is, when high-level protocol HLP opens a channel (session) via low-level protocol LLP, it identifies its peer according to LLP's definition of addresses. Third, while all protocols define what they expect in the form of a demux key, only a few protocols are in the business of defining network addresses for hosts. (IP is the only such protocol in the Internet protocol suite.) Thus, so as not to overly restrict the type of addresses accepted by any one protocol—thereby making it possible to flexibly configure different protocols on top of each other—the x-kernel allows protocols to *under specify* the form of addresses they accept.

Practically speaking, this means that at the top of the protocol graph, the application identifies each participant by first pushing a host address onto the participant stack, and then pushing a demux key that will be understood by the topmost protocol onto the stack. The topmost protocol then pops off the first component (the demux key), but only pops the host address off the stack if it is interested in defining its own notion of a host address. Suppose it is not. Then, the protocol pushes a new demux key onto the stack—one that is meaningful to the protocol upon which it depends—and opens that protocol. It never sees, nor cares about, the type of the host address. This happens all the way down the protocol graph—pop off the demux key and replace it with a demux key that the next lower level will understand—until a protocol that cares about host addresses is encountered. Such a protocol pops both the demux key **and** the host address off the address stack, translates the host address into some lower-level host address, and pushes this new host address, along with a new demux key, onto the stack.

In other words, no protocol is restricted to to being composed with protocols that expect a certain type of host address; that is, the protocol is under specified. Another way of saying this is that address stacks allow protocols to be *polymorphic* with respect to addresses it never examines.

4.2 Relative Protocol Numbers

It should be clear that the x-kernel was defined to allow a great deal of flexibility in how protocols are composed together; to avoid situations where protocol A must be composed on top of protocol B because A contains hard-coded dependencies on B. Being able to configure the protocol graph explicitly is one mechanism that advances this goal. The address stack mechanism just described is another. We now describe a third mechanism—how protocols specify their demux key.

In most conventional protocols, a low-level protocol uses a *relative* protocol number to identify the protocols above it. For example, IP identifies UDP with protocol number 17 and TCP with protocol number 6. It looks for these numbers in the header of an arriving packet to decide which high level protocol the packet should be passed to. These numbers are relative in that they only make sense to IP; if we were to configure a protocol graph in which UDP and TCP were above some other protocol, it could use some different set of numbers to decide which protocol should receive arriving packets.

The downside of relative addressing is that the low-level protocol needs to know about all the protocols that might be configured above it so that it can give them distinct relative protocol numbers. To circumvent this, protocols that have been especially designed to use the x-kernel use an *absolute* addressing scheme. The protocol number by which a given protocol is addressed is independent of which protocol is doing the addressing.

The x-kernel reconciles these two approaches by maintaining a table of relative protocol numbers. Rather than embed protocol numbers in the protocol source code, protocols learn the protocol numbers of protocols above them by querying this table using the following operation:

```
ProtId relProtNum(Prot hlp, Prot llp)
```

This operation returns the protocol number of the high-level protocol relative to the low-level protocol. This number will have to be cast into the appropriate type before it can be used.

The table of relative protocol numbers can then be configured with information that allows protocols to identify themselves according to the standard specifications, as well as to be more freely composed in non-standard ways. For example, the relative protocol number table might be configured as follows:

```
#
# prottbl
#
# This file describes absolute protocol id's
# and gives relative protocol numbers for those
# protocols which use them
#

eth         1
{
        ip          x0800
        arp         x0806
        rarp        x8035
        #
        # ethernet types x3*** are not reserved
        #
        blast       x3001
```

```
}
ip          2
{
        icmp       1
        tcp        6
        udp        17
        #
        # IP protocol numbers n, 91 < n < 255, are unassigned
        #
        blast      101
}
arp         3
rarp        4
udp         5
tcp         6
icmp        7
blast       8
rpc         9
```

This table specifies, for example, that IP is known by protocol number x0800 in the context of (relative to) protocol ETH, and that UDP is known by protocol number 17 relative to IP. Both of these examples are valid according to the standard specifications for ETH, IP, and UDP. Since the relative protocol numbers are carried in the packets, it is important to use the standard values if you want your implementation to interoperate with someone else's. Protocol BLAST is a non-standard protocol that one might want to configure on top of either ETH or IP; it is given protocol numbers x3001 and 101, respectively, in these two contexts.

Notice that in addition to giving certain relative protocol numbers, this table also assigns each protocol an absolute identifier; e.g., IP is given id 2 and BLAST is given id 8. For example, when protocol BLAST uses partRelProtNum to learn what protocol number to use to identify RPC, the operation returns 9; RPC's absolute protocol number. Unlike relative protocol numbers, absolute numbers are not transmitted in packets, so they need only be locally consistent.

5 Event Library

Another common activity for protocols is to schedule an event to happen some time in the future. To understand how a protocol might use such events, consider the situation where a network sometimes fails to deliver a message to the destination. A protocol that wants to offer a reliable channel across such a network might, after sending a message, schedule an event that is to occur after a certain period of time. If the protocol has not received confirmation that the message was delivered by the time this event happens, then the protocol retransmits the message. In this case, the event implements a *timeout*; we will see an example of a protocol that uses timeouts in a later section. Note that another use for events is to perform periodic maintenance functions, such as garbage collection.

This section introduces the interface to the x-kernel's event manager, which allows protocols to schedule a procedure that is to be called after a period of time. (See [6] for a description of the algorithm and data structures that underly the event manageer.)

The event manager defines a single object—the Event—and the following operation:

Event evSchedule(EvFunc function, void *argument, int time)

This operation schedules an event that executes the specified function with the given argument after a delay of time microseconds. (EvFunc is a pointer to function that returns a void.) A handle to the event is returned, and this can be used to cancel the event at some later time. When an event fires, a new process is created to run the specified function; that is, the event runs asynchronously with respect to the rest of the system. Since each event occurs at most one time, if the protocol wants a repeating event, then the next incarnation of the event should be re-scheduled using evSchedule as the last action taken in the event handling function.

A second operation:

```
EvCancelReturn evCancel(Event event)
```

is used to cancel the given Event. evCancel is called, for example, because a source has received confirmation that a message it sent was successfully delivered, and so there is no reason for it to retransmit the message. The operation's return value, as defined by the enumeration type EvCancelReturn, is set to EVENT_FINISHED if the event has already happened, EVENT_RUNNING if the event is currently running, and EVENT_CANCELLED if the event has not run and can be guaranteed to not run.

Finally, the following operation releases a handle to an event. As soon as function completes, the internal resources associated with the event are freed:

```
void evDetach(Event event)
```

6 Map Library

The final x-kernel tool we describe is the id mapper. This tool provides a facility for maintaining a set of bindings between identifiers. The id mapper supports operations for adding new bindings 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 (e.g., addresses, demultiplexing keys) into capabilities for (pointers to) x-kernel objects such as sessions. (See [1] for a description of how the map library is implemented.)

The id mapper supports two main objects: *maps* and *bindings*, represented by the types Map and Binding, respectively. A map is simply a table of bindings, where each binding is given by the pair $\langle$ external key, internal id $\rangle$. An external key is a variable length byte string, and an internal id is a fixed-sized identifier (e.g., a 32 or 64-bit memory address). Typically, an external key is constructed from various fields in a message header, and an internal id is a pointer to a protocol or session object. A map is created with the following operation:

```
Map mapCreate(int number, int size)
```

This operation creates a map that is able to hold number bindings in it, where the external keys bound in this map are size bytes long.

Once a map is created, protocols can perform two basic operations on it. The first puts bindings into the map and the latter resolves external keys according to the map and returns the corresponding internal id:

```
Binding mapBind(Map map, void *key, void *id)

XkReturn mapResolve(Map map, void *key, void **id)
```

The first operation inserts a binding of key to id into the specified map, and returns a pointer to the resulting binding. If key is already bound to some id in the map, then a pointer to that existing binding is returned. The second operation returns the internal id bound to the specified key in the given map. If the key is not found in the map, then mapResolve reports failure by returning XK_FAILURE.

The id mapper also provides a pair of operations for removing bindings from a map:

```
XkReturn mapRemoveBinding(Map map, Binding binding)

XkReturn mapRemoveKey(Map map, void *key)
```

The first removes the specified binding—the value returned by an earlier mapBind—and the second removes the binding for the specified key. Both operations return a failure code if the binding does not exist in the map.

Protocols generally maintain two maps: an active map and a passive map. Active maps are used to map keys found in incoming messages into the session that will process the message. Thus, the active map holds information about the set of currently active connections. Passive maps are used to bind keys found in incoming messages into protocol objects, thereby allowing a protocol to create a session when a message that is part of a new connection arrives.

Typically, a protocol binds an active key to a session in its implementation of xOpen, and a passive key to a protocol object in its xOpenEnable routine. These bindings are then used in the protocol's xDemux operation. This general pattern is illustrated in the example given Section 7.

7 Example Protocol

This section presents an example protocol—A Simple Protocol (ASP)—that illustrates how the interfaces and support routines described in previous sections are used. ASP is a complete, working protocol that is typically configured on top of IP in an x-kernel protocol graph. Because of its simplicity, ASP does not implement any interesting algorithms; it only serves to illustrate the boilerplate code that is common to all protocols. The next section illustrates some of the more complicated things that protocols do.

ASP supports an unreliable message delivery service, where the two end-points of an ASP channel are identified by a pair of *ports*. In the x-kernel, a session object implements each end-point of an ASP channel, with each session uniquely identified by the following 4-tuple:

⟨ local IP host, remote IP host, local ASP port, remote ASP port ⟩

In other words, ASP demultiplexes each incoming message to the appropriate session by using this 4-tuple as a demux key. This is ASP's only significant function—to add a multiplexing/demultiplexing function to the protocols below it.

ASP does very little processing on each message—it only checks the length of the received messages, and truncates the message to the specified length, if necessary. ASP is an unreliable protocol, in the sense that it adds no reliability to the protocols below it; i.e., it does not retransmit lost messages.

7.1 Header Files

By convention, the header information required by ASP is organized into two files: asp.h and asp_internal.h. File asp.h contains only definitions required by other protocols; protocols that depend on ASP must include asp.h in order to use these definitions. In the case of ASP, asp.h defines the ASP port to be an unsigned short (16-bit value).

```
/*
 * asp.h
 */

typedef u_short ASPport;
```

File asp_internal.h contains ASP-specific definitions that other protocols do not need to know about. For example, it defines the format of the ASP header (ASPhdr), protocol-specific state information (ProtlState), and session-specific state information (SessnState).

```
/*
 * asp_internal.h
 */
#include "xkernel.h"
#include "ip.h"
#include "asp.h"

/* ASP message header definition */

typedef struct header {
    ASPport sport;  /* source port       */
```

```
    ASPport dport;  /* destination port */
    u_short ulen;   /* ASP length       */
} ASPhdr;

#define HLEN (sizeof(ASPhdr))

/* protocol and session states */

typedef struct pstate {
    Map activemap;
    Map passivemap;
} ProtlState;

typedef struct sstate {
    ASPhdr hdr;
} SessnState;

/* active and passive maps */

typedef struct {
    Sessn   lls;
    ASPport localport;
    ASPport remoteport;
} ActiveId;

typedef ASPport PassiveId;

#define ACTIVE_MAP_SIZE  101
#define PASSIVE_MAP_SIZE 23

/* UPI function declarations */

void            asp_init(Protl);
static Sessn    aspOpen(Protl, Protl, Protl, Part *);
static XkReturn aspOpenEnable(Protl, Protl, Protl, Part *);
static XkReturn aspDemux(Protl, Sessn, Msg *);
static XkHandle aspPush(Sessn, Msg *);
static XkReturn aspPop(Sessn, Sessn, Msg *, void *);
static Sessn    aspCreateSessn(Protl, Protl, Protl, ActiveId *);
static XkReturn aspClose(Sessn);
static int      aspControlProtl(Protl, int, char *, int) ;
static int      aspControlSessn(Sessn, int, char *, int) ;
static Part     *aspGetParticipants(Sessn);

/* internal function declarations */

static void     getproc_protl(Protl);
static void     getproc_sessn(Sessn);
static long     aspHdrLoad(ASPhdr *, char *, long);
static void     aspHdrStore(ASPhdr *, char *, long);
```

```
/* trace variable used in conjunction with xTrace stmts */
int traceaspp;
```

One of the main definitions contained in asp_internal.h concerns the demultiplexing function. In particular, the state associated with the ASP protocol object includes two maps: activemap and passivemap. The former map is used by ASP's demux routine to map incoming messages to an existing session. It uses ActiveId as the demultiplexing key. Although conceptually the active map key is given by the 4-tuple described above, in practice, we take advantage of the fact that the underlying IP session already corresponds to a local/remote host pair, and we use this session object in lieu of these two IP addresses.

The second map—passivemap—is used to record high-level protocols that have done a passive open on ASP. In this case, the corresponding key that is used to search this map, called the PassiveId, is given by the ASP port on which the local protocol has done an xOpenEnable.

The session state contains a header template which is copied to the front of outgoing messages. The ASP message header consists of local and remote port numbers, and a field indicating the length of the data message plus ASP header.

7.2 Initialization

The C functions that implement a given protocol can be distributed across multiple .c files. In the case of ASP, they are all contained in a single file named asp.c. The rest of this section walks through these functions one at at time.

We begin with function asp_init, which is called at system startup time. This function initializes the ASP protocol object, including protocol state and maps. It then does an xOpenEnable call on the protocol configured below it (usually IP) to inform it that ASP is willing to accept messages from any host. xGetDown is used to obtain a handle for the protocol that was configured below ASP. (In theory, ASP may have more than one protocol configured below it; the second argument to xGetDown asks for the first of these.)

```
void
asp_init(Protl self)
{
    ProtlState *pstate;
    Protl      llp;
    Part       part;

    getproc_protl(self);

    /* create and initialize protocol state */
    pstate = X_NEW(ProtlState);
    bzero((char *)pstate, sizeof(ProtlState));
    self->state = (void *)pstate;
    pstate->activemap  = mapCreate(ACTIVE_MAP_SIZE, sizeof(ActiveId));
    pstate->passivemap = mapCreate(PASSIVE_MAP_SIZE, sizeof(PassiveId));

    /* find lower level protocol and do a passive open on it */
    llp = xGetProtlDown(self, 0);
    if (!xIsProtl(llp))
        Kabort("ASP could not get lower protocol");
    partInit(&part, 1);
    partPush(part, ANY_HOST, 0);
    if (xOpenEnable(self, self, llp, &part) == XK_FAILURE) {
        xTrace0(aspp, TR_ALWAYS,
               "asp_init: openenable on lower protocol failed");
        xFree((char *) pstate);
```

```
        return;
    }
}
```

Because in this example we are dealing not just with the interfaces between objects but with their implementation, we need to deal with some of the internal structure of the Protl and Sessn objects, which we have glossed over up until now. For example, in the preceding code, we initialized the state data structure in the ASP Protl object. One important part of initialization of the Protl object is to fill in the table that will cause operations on the object (e.g., xOpen) to invoke the appropriate function to implement that operation (e.g., aspOpen). In the case of ASP, the subroutine getproc_protl fills out this operation table:

```
static void
getproc_protl(Protl p)
{
    /* fill in the function pointers to implement protocol operations */
    p->open         = aspOpen;
    p->openenable   = aspOpenEnable;
    p->demux        = aspDemux;
    p->controlprotl = aspControlProtl;
}
```

7.3 Opening Connections

When a high level protocol wishes to establish an ASP channel to a remote host, it will call xOpen, which causes aspOpen to be called. This function extracts the local and remote ASP ports from the participant list passed down by the high level protocol, and passes the resulting participant list, via another xOpen call, down to IP. A new ASP session (asp_s) is then created to handle messages sent and received on this channel. If a high level protocol attempts to re-open a channel—i.e., one that is found in the active map—the open fails; otherwise, the newly created session is returned to the calling protocol.

Notice that aspOpen uses the x-kernel operation partPop to extract address information from the participant list; there is also a partPush operation that is used to attach addressing information to the participant list. Briefly, the participant list is an array of addresses, where by convention, the first element of this array identifies the remote participant and the second element identifies the local participant. Each element in the array is, in turn, represented by a stack of addresses; hence "push" and "pop" in the operation names. It is a stack rather than a flat structure because a given participant might be identified with a multi-component address, for example, a host address and a port number. In this example, aspOpen extracts the port number from each stack, but leaves the host number on the stack for some lower level protocol (e.g., IP) to process.

```
static Sessn
aspOpen(self, hlp, hlpType, p)
Protl self, hlp, hlpType;
Part  *p;
{
    ActiveId   key;
    Sessn      asp_s, lls;
    ProtlState *pstate = (ProtlState *)self->state;

    bzero((char *)&key, sizeof(key));

    /* high level protocol must specify both local and remote ASP port */
    key.remoteport = *((ASPport *)partPop(p[0]));
    key.localport  = *((ASPport *)partPop(p[1]));
```

```
    /* attempt to open session on protocol below this one */
    lls = xOpen(self, self, xGetProtlDown(self, 0), p);
    if (lls != ERR_SESSN) {
        key.lls = lls;
        /* check for this session in the active map */
        if (mapResolve(pstate->activemap, &key, (void **)&asp_s) == XK_FAILURE){
            /* session wasn't already in map, so initialize it */
            asp_s = aspCreateSessn(self, hlp, hlpType, &key);
            if (asp_s != ERR_SESSN)    /* A successful open! */
                return asp_s;
        }
        /* if control makes it this far, an error has occurred */
        xClose(lls);
    }
    return ERR_SESSN;
}
```

Note that the real work of initializing the session is performed by asp_init_session, presented below. Before we get into the details of that routine, however, we'll take a look at aspOpenEnable. This is the routine that is called indirectly through xOpenEnable by a high level protocol that wants to do a passive open on ASP. The high level protocol specifies its willingness to receive messages on a specific ASP port, and aspOpenEnable records this fact in its passive map. Note that the x-kernel defines an Enable object that is used by aspOpenEnable to record the necessary information about the passive open, including a reference count (rcnt) of how many times that particular port has been enabled. Also, keep in mind that a session object is not created at this time; it will be created in aspDemux when a message arrives addressed to that port.

```
static XkReturn
aspOpenEnable(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    PassiveId  key;
    ProtlState *pstate = (ProtlState *)self->state;
    Enable     *e;

    key = *((ASPport *)partPop(*p));

    /* check if this port has already been openenabled */
    if (mapResolve(pstate->passivemap, &key, (void **)&e) != XK_FAILURE) {
        if (e->hlp == hlp) {
           /* this port was openenabled previously by the same hlp */
            e->rcnt++;
            return XK_SUCCESS;
        }
        /* this port was openenabled previously by a different hlp - error */
        return XK_FAILURE;
    }

    /* this will be a new enabling, so create and initialize Enable object, */
    /* and enter the binding of port/enable object in the passive map */
    e = X_NEW(Enable);
    e->hlp     = hlp;
    e->hlpType = hlpType;
```

```
    e->rcnt    = 1;
    e->binding = mapBind(pstate->passivemap, &key, e);
    if (e->binding == ERR_BIND) {
        xFree((char *)e);
        return XK_FAILURE;
    }
    return XK_SUCCESS;
}
```

Finally, the actual work of initializing an ASP session is performed in the subroutine aspCreateSessn given below. This subroutine calls an x-kernel operation—xCreateSessn—to allocate a new session object and connect it to other protocol and session objects in the appropriate way; e.g., the new session points to the high level protocol that created it (hlp) and the low-level session that it will use to send and receive messages (lls). In addition, the new session object will need to know where to find the routines that implement the various operations it supports, such as xPush and xPop. getproc_sessn (presented below) is the routine that fills in the ASP-specific function pointers for these operations; it is called by xCreateSessn.

Note that aspCreateSessn is not only called from aspOpen, but also from aspDemux (given below) when it needs to create a new session as a result of an earlier passive open. Also notice that in the case of aspCreateSessn, the only protocol-specific initialization required is to set up a header template asph that can later be prepended to the front of outgoing messages.

```
static Sessn
aspCreateSessn(Protl self, Protl hlp, Protl hlpType, ActiveId *key)
{
    Sessn      s;
    ProtlState *pstate = (ProtlState *)self->state;
    SessnState *sstate;
    ASPhdr     *asph;

    /* create the session object and initialize it */
    s = xCreateSessn(getproc_sessn, hlp, hlpType, self, 1, &key->lls);
    s->binding = mapBind(pstate->activemap, key, s);
    sstate = X_NEW(SessnState);
    s->state = (char *)sstate;

    /* create an ASP header */
    asph = &(sstate->hdr);
    asph->sport = key->localport;
    asph->dport = key->remoteport;
    asph->ulen  = 0;

    return s;
}

static void
getproc_sessn(Sessn s)
{
    /* fill in the function pointers to implement session operations */
    s->push            = aspPush;
    s->pop             = aspPop;
    s->controlsessn    = aspControlSessn;
    s->getparticipants = aspGetParticipants;
```

```
    s->close            = aspClose;
}
```

7.4 Demultiplexing Incoming Messages

Function aspDemux is called indirectly through xDemux by an IP session to pass an incoming message up to ASP. (More generally, it could be called by any low level protocol configured below ASP, but this will usually be IP.) This function extracts the header information from the message, and then demultiplexes the message to the appropriate session as follows. First, aspDemux consults the protocol's active map to see if an existing session exists for the ASP channel to which the message belongs. If such a session exists, the message is passed to that session via the xPop operation. If it does not exist, aspDemux then checks the passive map to see if the message is addressed to a port on which a high level session has performed an *xOpenEnable*. If the port is found in the passive map, a new session is created—by invoking aspCreateSessn—and the message is dispatched to that session. If an appropriate entry is not found in either the active or passive maps, the message is discarded.

```
static XkReturn
aspDemux(Protl self, Sessn lls, Msg *dg)
{
    char       *buf;
    ASPhdr     h;
    ActiveId   activeid;
    ProtlState *pstate = (ProtlState *)self->state;
    Sessn      s;
    PassiveId  passiveid;
    Enable     *e;

    /* extract the header from the message */
    buf = msgPop(dg, HLEN);
    if (buf == NULL)
        return XK_FAILURE;
    aspHdrLoad(&h, buf, HLEN);

    /* construct a demux key from the header */
    bzero((char *)&activeid, sizeof(activeid));
    activeid.lls        = lls;
    activeid.localport  = h.dport;
    activeid.remoteport = h.sport;

    /* see if demux key is in the active map */
    if (mapResolve(pstate->activemap, &activeid, (void **)&s) == XK_FAILURE) {
        /* didn't find an active session, so check passive map */
        passiveid = h.dport;
        if (mapResolve(pstate->passivemap, &passiveid, (void **)&e) ==
                XK_FAILURE) {
            /* drop the message */
            return XK_SUCCESS;
        }

        /* port was enabled, so create a new session and inform hlp */
        s = aspCreateSessn(self, e->hlp, e->hlpType, &activeid);
        if (s == ERR_SESSN)
```

```
            return XK_SUCCESS;
        xDuplicate(lls);
        xOpenDone(e->hlp, self, s);
    }

    /* found (or created) an appropriate session, so pop to it */
    return xPop(s, lls, dg, &h);
}
```

The call to xDuplicate near the end of aspDemux is necessary to increment the reference count on the newly created session. More on reference counts in Section 8.

Subroutine aspHdrLoad is responsible for converting the fields in the ASP header from network byte order into the local host's byte order. It uses the library routine ntohs, which stand for network-to-host-short. A counterpart routine—aspHdrStore—performs the opposite conversion; it translates host byte order to network byte order. This latter routine is used when preparing a message for transmission, as discussed in the next subsection. Notice that both routines simultaneously swap bytes and copy the header between the message and the template.

```
static long
aspHdrLoad(ASPhdr *hdr, char *src, long len)
{
    /* copy from src to hdr, then convert network byte order to hsot order */
    bcopy(src, (char *)hdr, HLEN);
    hdr->ulen  = ntohs(hdr->ulen);
    hdr->sport = ntohs(hdr->sport);
    hdr->dport = ntohs(hdr->dport);
    return HLEN;
}

static void
aspHdrStore(ASPhdr *hdr, char *dst, long len)
{
    /* convert host byte order to network order, then copy from hdr to dst */
    /* (note: argument 'hdr' is changed by the following code) */
    hdr->ulen  = htons(hdr->ulen);
    hdr->sport = htons(hdr->sport);
    hdr->dport = htons(hdr->dport);
    bcopy((char *)hdr, dst, HLEN);
}
```

7.5 Sending and Receiving Messages

We now turn our attention to the ASP-specific implementation of xPush and xPop—aspPush and aspPop—the two routines that embody the semantics of ASP. Because of its simple nature, aspPush simply prepends a copy of the header stored in the session state onto the outgoing message, and passes the resulting message to the IP session beneath it. The header template found in the session state already contains the appropriate local and remote ASP ports, since these will not change in the lifetime of a session. However, the message length must be calculated and inserted into the appropriate header field. Subroutine aspHdrStore, as defined in the previous subsection, converts this header template from host byte order to network byte order.

```
static XkHandle
aspPush(Sessn self, Msg *msg)
{
```

```
    SessnState *sstate = (SessnState *)self->state;
    ASPhdr     hdr;
    char       *buf;

    /* create a header by inserting length into header template */
    hdr = sstate->hdr;
    hdr.ulen = msgLength(msg) + HLEN;

    /* attach header to message and pass it on down the stack */
    buf = msgPush(msg, HLEN);
    aspHdrStore(&hdr, buf, HLEN);
    return xPush(xGetSessnDown(self, 0), msg);
}
```

Again because of its simplicity, the only work that aspPop performs is to check the length field in the message header, and truncate the message, if necessary. Notice that the last thing aspPop does is invoke xDemux on the high-level protocol that had earlier opened the session.

```
static XkReturn
aspPop(Sessn self, Sessn lls, Msg *msg, void *hdr)
{
    ASPhdr *h = (ASPhdr *)hdr;

    /* truncate message to length shown in header */
    if (h->ulen - HLEN < msgLength(msg))
        msgTruncate(msg, (int)h->ulen);

    /* pass the message to the next protocol up the stack */
    return xDemux(xGetUp(self), self, msg);
}
```

7.6 Control Operations

High-level protocols and sessions may request information from or change the state of ASP protocol and session objects by calling the following two control functions. If the opcode used in the call is not recognized by the ASP control function, one of IP's control functions is called with the same opcode. In these routines checkLen is a simple macro that makes sure the the buffer len is large enough to hold the value being returned.

```
static int
aspControlProtl(Protl self, int opcode, char *buf, int len)
{
    switch (opcode) {
        case GETMAXPACKET:
        case GETOPTPACKET:
            checkLen(len, sizeof(int));
            if (xControlProtl(xGetProtlDown(self, 0), opcode, buf, len) <
                    sizeof(int))
                return -1;
            *(int *)buf -= HLEN;
            return sizeof(int);
        default:
            return xControlProtl(xGetProtlDown(self, 0), opcode, buf, len);
```

```
    }
}

static int
aspControlSessn(Sessn self, int opcode, char *buf, int len)
{
    SessnState *sstate = (SessnState *)self->state;
    ASPhdr     *hdr;

    hdr = &(sstate->hdr);
    switch (opcode) {
        case GETMYPROTO:
            checkLen(len, sizeof(long));
            *(long *)buf = sstate->hdr.sport;
            return sizeof(long);
        case GETPEERPROTO:
            checkLen(len, sizeof(long));
            *(long *)buf = sstate->hdr.dport;
            return sizeof(long);
        case GETMAXPACKET:
        case GETOPTPACKET:
            checkLen(len, sizeof(int));
            if (xControlSessn(xGetSessnDown(self, 0), opcode, buf, len) <
                    sizeof(int))
                return -1;
            *(int *)buf -= HLEN;
            return sizeof(int);
        default:
            return xControlSessn(xGetSessnDown(self, 0), opcode, buf, len);
    }
}
```

In addition to invoking these control operations on low-level protocols and sessions, high-level entities can query ASP for participants associated with the session. This is done by invoking xGetParticipants on the session. The ASP-specific implementation of this operation turns around and asks the session below it (i.e., an IP session) for the same information.

```
static Part *
aspGetParticipants(Sessn self)
{
    Part       *p;
    int        numParts;
    SessnState *sstate = (SessnState *)self->state;
    long       localPort, remotePort;

    p = xGetParticipants(xGetSessnDown(self, 0));
    if (!p)
        return NULL;
    numParts = partLength(p);
    if (numParts > 0 && numParts <= 2) {
        if (numParts == 2) {
            localPort = (long)sstate->hdr.sport;
```

```
            partPush(p[1], (void *)&localPort, sizeof(long));
        }
        remotePort = (long)sstate->hdr.dport;
        partPush(p[0], (void *)&remotePort, sizeof(long));
        return p;
    }
    else    /* Bad number of participants */
        return NULL;
}
```

7.7 Close

Finally, aspClose is called indirectly through xClose when a high-level protocol wants to close a session it had opened earlier. This routine removes the entry in the active map corresponding to this session, closes the lower level session, and destroys the session object. As you can see, the session object includes a field in which the binding in the active map for this session was recorded (binding). Also, the routine xMyProtl returns the protocol object that corresponds to this session (e.g., the protocol object that represents ASP).

```
static XkReturn
aspClose(Sessn s)
{
    ProtlState *pstate = (ProtlState *)xMyProtl(s)->state;

    /* remove this session from the active map */
    mapRemoveBinding(pstate->activemap, s->binding);

    /* close the lower level session on which it depends */
    xClose(xGetSessnDown(s, 0));

    /* de-allocate the session object itself */
    xDestroySessn(s);

    return XK_SUCCESS;
}
```

8 Reference Counting Sessions

The *x*-kernel maintains reference counts for sessions in order to facilitate their destruction. This section discusses the system support for reference counts. We discuss reference counts in the context of sessions (rather than protocols) because protocols are relatively static objects and the issues surrounding their reference counts are not very interesting.

Constructing a protocol that directly manipulates reference counts and does so correctly can be awkward and tedious. The *x*-kernel has addressed this difficulty by: (1) moving all direct reference count manipulation into the *x*-kernel infrastructure; (2) explicitly documenting how references to other sessions may be used; and (3) adding system support for session caching and garbage collection.

8.1 References

A Sessn reference is a pointer to a Sessn. References come in two flavors: *permanent* and *temporary*. A permanent reference can be used indefinitely. As long as the holder of a permanent reference does not call xClose on the reference, the holder knows that the pointer will remain valid. Permanent references are returned by xOpen. A Sessn pointer received as a parameter in a function call is a temporary reference. It can only be safely used for the duration of that function call.

A permanent reference may be created from a temporary reference by using the following operation on session:

```
XkReturn xDuplicate(Sessn session)
```

For example, xDemux passes a reference to the lower session as a parameter to the upper protocol. If the upper protocol wishes to save this reference beyond the extent of the call, the protocol should call xDuplicate on the session. The protocol can then safely use the reference until it calls xClose.

8.2 Reference counts

Session reference counts are a sum of:

- the number of permanent external references to the session
- the number of outstanding xPop's on the session

8.2.1 Counting External References

Several of the UPI functions are involved in maintaining the count of permanent external references. They are listed here along with their semantics with respect to reference counts.

xCreateSessn(...)

> The newly created session has an initial reference count of 0.

xOpen(...)

> The session returned by invoking the lower protocol's open routine has its reference count incremented before it is returned to the caller of xOpen, indicating that the caller now has a permanent reference to the session.

xClose(session)

> Decrements the reference count of the session, indicating that a permanent reference to the session has been released. The lower protocol's close operation is called *only* if the session's new reference count is zero.

xDuplicate(session)

> By default, increments the session's reference count, indicating that a new permanent reference to the session has been created. If the session has its own duplicate function, that is called instead.

8.2.2 Counting xPops

To understand the second component of the reference count, the number of outstanding xPops on a session, it is important to realize that a protocol does not have explicit references to its own sessions. That is, while a protocol usually maintains pointers to its sessions (in a map) and invokes operations on them, the sessions' reference counts do not take these pointers into account. (This is why xCreateSessn returns an object with a reference count of 0 and not 1.)

For a protocol to safely send this session pointer outside of the protocol (e.g., as a parameter in xDemux), it needs to turn the pointer into a reference by incrementing the session's reference count. It could do this by calling xDuplicate on the session as soon as it is extracted from the session map, making the external call, and then closing the session.

But since all protocols must do this for incoming messages, this functionality has been absorbed into xPop, that is, xPop increments the reference count of the session before calling the pop function and decrements it (possibly calling the session's close operation) afterwards.

It could be argued that all UPI functions, not just xPop, should indicate their use of a session by incrementing the reference count at their start and decrementing it at their completion. To perform any other UPI operation on a session, however, you must already have a reference to that session. As long as an xClose is not performed on that reference, the session is not going to go away; maintaining reference counts for these operations is not necessary. xPops are different in that a session's reference count does not reflect that its protocol may send messages up through it.

Note that reference counts will not help a protocol which performs an xClose on a session reference while another thread has an outstanding operation on that same reference. To perform such a sequence is a protocol error. If two threads share a session reference, they should either synchronize to avoid such sequences or they should duplicate the reference with xDuplicate and each thread should xClose its reference when it is through.

8.3 Internal vs. External Reference Counts

Session reference counts are meant to count the number of external references to the session (i.e., references held outside of the protocol). If a protocol must keep track of session references internal to the protocol itself, a separate mechanism must be used.

This requirement is driven by upper protocols that require that when they release all of their references to a lower session, then the session's reference count goes to zero. Virtual protocols that manipulate lower session's upper protocol pointers are specific examples of these types of upper protocols.

8.4 Session Caching Strategies

In the presence of correct management of reference counts, protocols may want to implement some sort of session caching. For example, many of the sessions created on the receiving end of a remote procedure call will have their reference counts go to zero after the reply has been sent, and without caching, all of those sessions will be destroyed. While caching may not be important for all protocols, it is vital to the performance of protocols that experience a high frequency of traffic from the same sources.

If a protocol does cache idle sessions, it is important to make the caching transparent to upper protocols (i.e., an upper protocol should not be able to distinguish a newly created lower session from a cached lower session that is being reused). One aspect of this requirement is that protocols that cache sessions must be sure to check for the presence of openEnables when reusing sessions which would otherwise be passively created, and to call xOpenDone when reusing such sessions from the cache.

Following are some examples of caching strategies and how a protocol might implement them. These are just examples; other strategies are certainly possible.

- Destroy the session if inactive.

  In this "null caching" strategy, the session's close routine destroys the session.

```
fooClose(s)
    xAssert(s->rcnt == 0);
    xDestroy(s);
```

- Garbage collection on a per-session basis.

  A garbage collector event is started for each idle session. When the event expires, the session is collected.

```
fooDemux()
    if (active session does not exist) {
        if (openEnable exists) {
```

```
            if (cached session exists) {
                evCancel(sstate->gc);
                sstate->gc = 0;
            }
            else
                s = fooOpen();
            xOpenDone(s, ...);
        }
        else {
            drop packet;
            return;
        }
    }
    xPop()

fooOpen()
    ...
    if (active session does not exist) {
        if (cached session exists) {
            evCancel(sstate->gc);
            sstate->gc = 0;
        }
    }
    ...

fooClose(s)
    sstate->gc = evRegister(foo_gc, s);
    mark session as cached

foo_gc(s)
    xDestroy(s);
```

- Garbage collection on a per-protocol basis.

  There is a single garbage collector for the protocol. It runs every so often and looks for idle sessions in the protocol's session map, calling a protocol-specific destroy function if it finds one.

  The *x*-kernel has a default session garbage collector that a protocol can use for this purpose. See xkernel/include/gc.h for the interface.

```
foo_init()
    ...
    initSessionCollector(cacheMap, interval, fooDestroy);

fooDemux()
    if (active session does not exist) {
        if (openEnable exists) {
           if (cached session exists) {
               move sessn from cached map to active map
           }
           else
               s = fooOpen();
```

```
                xOpenDone(s, ...);
            }
            else {
                drop packet;
                return;
            }
        }
        xPop()

    fooOpen()
        ...
        if (active session does not exist) {
            if (cached session exists) {
                move sessn from cached map to active map
            }
        }
        ...

    fooClose(s)
        xAssert(s->rcnt == 0);
        move sessn from active map to cached map

    fooDestroy(s)
        xDestroy(s);
```

9 More Example Protocols

This section shows code fragments from several different protocols, with the goal of illustrating some of the more common protocol algorithms. All of these examples are taken from protocols distributed with the *x*-kernel. See the actual source code for the complete protocol associated with each fragment.

9.1 Fragmentation

BLAST is an *x*-kernel protocol that fragments large messages into MTU-sized packets at the sender, and reassembles the fragments back into the complete message at the receiver. The following shows the implementation of blastPush, which contains the for loop that generates and transmits all the fragments in a message. Notice that it is not necessary to calculate how long the last fragment is because the msgBreak operation automatically makes each fragment the minimum of the specified size (FRAGMENT_SIZE) and how ever many bytes are left in the message.

```
static XkReturn
blastPush(Sessn s, Msg *msg)
{
    BlastState *state = s->state;
    BlastHdr   *hdr;
    int        num_frags, i;
    Msg        *fragment;
    char       *buf;

    /* get header template and set MID, incrementing last value used */
    hdr = state->hdr_template;
```

```
    if (state->mid == MAX_SEQ_NUM)
        state->mid = 0;
    hdr->MID = ++state->mid;

    /* determine number of fragments */
    if (msgLength(msg) <= FRAGMENT_SIZE)
        num_frags = 1;
    else
        num_frags = (msgLength(msg) + FRAGMENT_SIZE - 1)/FRAGMENT_SIZE;
    hdr->NumFrags = num_frags;

    /* create and transmit individual fragments */
    for (i = 1; i <= num_frags; i++) {
        /* carve a fragment off of original msg */
        msgConstructEmpty(fragment);
        msgBreak(msg, fragment, FRAGMENT_SIZE);

        /* fill in dynamic parts of header */
        hdr->len = msgLength(fragment);
        set_fragment_mask(hdr->mask, i);

        /* add header and send fragment */
        buf = msgPush(fragment, HDR_LEN);
        blast_hdr_store(hdr, buf, HDR_LEN, fragment);
        xPush(xGetDown(s, 0), fragment);

        /* save copy of fragment for future retransmit */
        save_for_retransmit(state->frag_list, fragment, i);
    }
    /* schedule DONE timer */
    state->event = evSchedule(giveup, 0, DONE);
}
```

This routine also uses the event library to schedule a timeout. We show a more complete example of how timeouts are implemented in a later example.

9.2 Reassembly

This example shows how IP reassembles fragments that arrive over the network back into a complete datagram. One reason we give this particular piece of code is that it is representative of a large fraction of networking code—it does little more than tedious and unglamorous bookkeeping. In this case, the code has to keep track of what fragments have and have not arrived.

First, we define the key data structure (FragList) that is used to hold the individual fragments that arrive at the destination. Incoming fragments are saved in this data structure until all the fragments in the original datagram have arrived, at which time they are reassembled into a complete datagram and passed up to some higher level protocol. Note that each element in FragList contains either a fragment or a hole.

```
#define FRAGOFFMASK      0x1fff
#define FRAGOFFSET(flag) ((fragflag) & FRAGOFFMASK)
#define INFINITE_OFFSET  0xffff

typedef struct fid {
```

```
    IpHost  source;
    IpHost  dest;
    u_char  prot;
    u_char  pad;
    u_short ident;
} FragId;

typedef struct hole {
    u_int first;
    u_int last;
} Hole;

#define HOLE  1
#define FRAG  2

typedef struct fragif {
    u_char type;
    union {
        Hole hole;
        Msg  frag;
    } u;
    struct fragif *next, *prev;
} FragInfo;

typedef struct FragList {
    u_short  nholes;
    FragInfo head;       /* dummy header node */
    Bind     binding;
    bool     gcMark;
} FragList;
```

The reassembly routine, ipReassemble, takes the session to which the datagram belongs (s), an incoming datagram (dg), and the IP header for that datagram (hdr) as arguments. The final argument (fragMap) is used to map the incoming datagram into appropriate FragList. (Recall that the group of fragments that are being reassembled together are uniquely identified by several fields in the IP header, as defined by structure FragId given above.)

The actual work done in ipReassemble is straightforward; as stated above, it is mostly bookkeeping. First, the routine extracts the fields from the IP header that uniquely identify the datagram being reassembled, constructs a key from these fields, and looks this key up in fragMap to find the appropriate FragList. Next, it inserts the new fragment into this FragList. This involves comparing the sum of the offset and length of this fragment with the offset of the next fragment in the list. Some of this work is done in subroutine hole_create, which is given below. Finally, ipReassemble checks to see if all the holes are filled. If all the fragments are present, it calls the x-kernel routine msgJoin to actually reassemble the fragments into a whole datagram and then calls xDemux to pass this datagram up the protocol graph.

```
static XkReturn
ipReassemble(Sessn s, Msg *dg, IpHdr *hdr, Map fragMap)
{
    FragId   fragid;
    FragList *list;
    FragInfo *fi, *prev;
    Hole     *hole;
    u_short  offset, len;
```

```
/* extract fragmentation info from header and create id for this frag */
offset = FRAGOFFSET(hdr->frag)*8;
len = hdr->dlen - GET_HLEN(hdr) * 4;
bzero((char *)&fragid, sizeof(FragId));
fragid.source = hdr->source;
fragid.dest   = hdr->dest;
fragid.prot   = hdr->prot;
fragid.ident  = hdr->ident;

/* find reassembly list for this frag; create one if this none exists */
if (mapResolve(fragMap, &fragid, (void **)&list) == XK_FAILURE) {
    list = X_NEW(FragList);
    list->binding = mapBind(fragMap, &fragid, list);

    /* initialize list with a single hole spanning the whole datagram */
    list->nholes     = 1;
    list->head.next  = fi = X_NEW(FragInfo);
    fi->next         = 0;
    fi->type         = HOLE;
    fi->u.hole.first = 0;
    fi->u.hole.last  = INFINITE_OFFSET;
}
list->gcMark = FALSE;

prev = &list->head;
for (fi = prev->next; fi != 0; prev = fi, fi = fi->next) {
    if (fi->type == FRAG)
        continue;
    hole = &fi->u.hole;
    if (offset < hole->last && offset + len > hole->first) {
        /* check to see if frag overlaps previously received frags */
        if (offset < hole->first) {
            /* truncate message from left */
            msgPop(dg, hole->first - offset);
            offset = hole->first;
        }
        if (offset + len > hole->last) {
            /* truncate message from right */
            msgTruncate(dg, hole->last - offset);
            len = hole->last - offset;
        }

        /* now check to see if new hole(s) need to be made */
        if (offset + len < hole->last && hdr->frag & MOREFRAGMENTS) {
            /* creating new hole above */
            hole_create(prev, fi, (offset+len), hole->last);
            list->nholes++;
        }
        if (offset > hole->first) {
            /* creating new hole below */
```

```
                hole_create(fi, fi->next, hole->first, (offset));
                list->nholes++;
            }

            /* change this FragInfo structure to be an FRAG */
            list->nholes--;
            fi->type = FRAG;
            msgConstructCopy(&fi->u.frag, dg);
            break;
        } /* if found a hole */
    } /* for loop */

    /* check to see if we're done, and if so, pass datagram up */
    if (list->nholes == 0)
    {
        Msg fullMsg;

        /* now have a full datagram */
        msgConstructEmpty(&fullMsg);
        for (fi = list->head.next; fi != 0; fi = fi->next)
            msgJoin(&fullMsg, &fi->u.frag, &fullMsg);
        mapRemoveBinding(fragMap, list->binding);
        ipFreeFragList(list);
        xDemux(xGetUp(s), s, &fullMsg);
        msgDestroy(&fullMsg);
    }
    return XK_SUCCESS;
}
```

Subroutine hole_create creates a new hole in the fragment list that begins at offset first and continues to offset last. It makes use of the x-kernel utility X_NEW, which creates an instance of the given structure.

```
static int
hole_create(FragInfo *prev, FragInfo *next, u_int first, u_int last)
{
    FragInfo *fi;

    /* creating new hole from first to last */
    fi = X_NEW(FragInfo);
    fi->type         = HOLE;
    fi->u.hole.first = first;
    fi->u.hole.last  = last;
    fi->next         = next;
    prev->next       = fi;
}
```

Finally, note that these routines do not capture the entire picture of reassembly. What is not shown is a background process (x-kernel event) that periodically checks to see if there has been any recent activity on this datagram (it looks at field gcMark), and if not, deletes the corresponding FragList. IP does not attempt to recover from the situation where one or more of the fragments does not arrive; it simply gives up and reclaims the memory that was being used for reassembly.

9.3 Synchrony, Timeouts, and Blocking

We now show an example of a hybrid protocol, named CHAN, that turns an underlying asynchronous communication service into a synchronous communication service. That is, CHAN supports the xPush/xDemux/xPop from below, and the xCall/xCallDemux/xCallPop from above. CHAN is also interesting because it illustrates how timeouts are scheduled in the *x*-kernel, and how how a protocol blocks, waiting for a reply message.

CHAN supports request/reply channels between a pair of hosts. The client sends a request message on a channel and blocks waiting for a reply message. The server accepts the request message and responds with a reply message. The protocol defines two key data structures: ChanHdr and ChanState. Both of these data structures are defined in a private .h file (e.g., chan_internal.h), rather than the chan.h file included by other protocols.

The fields in ChanHdr are fairly straightforward. The Type field specifies the type of the message; in this case, the possible types are REQ, REP, ACK and PROBE. The ProtNum field identifies the high-level protocol that depends on CHAN. The CID field uniquely identifies the logical channel to which this message belongs. This is a 16-bit field, meaning that CHAN supports up to 64K concurrent request/reply transactions between any pair of hosts. The MID field uniquely identifies each request/reply pair; the reply message has the same MID as the request. Finally, the BID field gives the *boot id* for the host. A machine's boot id is a number that is incremented each time the machine reboots; this number is read from disk, incremented, and written back to disk during the machine's startup procedure. This number is then put in every message sent by that host.

The fields in ChanState will be explained by the code that follows. The one thing to note at this point is that ChanState includes a hdr_template field, which is a copy of the CHAN header. Many of the fields in the CHAN header remain the same for all messages sent out over this channel. These fields are filled in when the channel is created (not shown); only the fields that change are modified before a given message is transmitted.

```
typedef struct {
    u_short Type;       /* message type: REQ, REP, ACK, PROBE */
    u_short CID;        /* unique channel id */
    int     MID;        /* unique message id */
    int     BID;        /* unique boot id */
    int     Length;     /* length of message */
    int     ProtNum;    /* high-level protocol number */
} ChanHdr;

typedef struct {
    u_char    type;             /* type of session: CLIENT or SERVER */
    u_char    status;           /* status of session: BUSY or IDLE */
    Event     event;            /* place to save timeout event */
    int       retries;          /* number of times retransmitted */
    int       timeout;          /* timeout value */
    XkReturn  ret_val;          /* place to save return value */
    Msg       *request;         /* place to save request message */
    Msg       *reply;           /* place to save reply message */
    Semaphore reply_sem;        /* semaphore the client blocks on */
    int       mid;              /* message id for this channel */
    int       bid;              /* boot id for this channel */
    ChanHdr   hdr_template;     /* header template for this channel */
} ChanState;
```

The CHAN-specific implementation of xCall is given by the following routine, named chanCall. The first thing to notice is that ChanState includes a field named status that indicates whether or not this channel is being used. If the channel is currently in use, then chanCall returns failure.

The next thing to notice about chanCall is that after filling out the message header and transmitting the request message, the calling process is blocked on a semaphore (reply_sem); semWait is the x-kernel semaphore operation introduced in Section 2. When the reply message eventually arrives, it is processed by CHAN's xPop routine (see below), which copies the reply message into state variable reply, and signals this blocked process. The process then returns. Should the reply message not arrive, then timeout routine retransmit is called (see below). This event is scheduled in the body of chanCall.

```
static XkReturn
chanCall(Sessn self, Msg *msg, Msg *rmsg)
{
    ChanState *state = (ChanState *)self->state;
    ChanHdr   *hdr;
    char      *buf;

    /* ensure only one transaction per channel */
    if (state->status != IDLE)
        return XK_FAILURE;
    state->status = BUSY;

    /* save a copy of request msg and pointer to reply msg*/
    msgConstructCopy(&state->request, msg);
    state->reply = rmsg;

    /* fill out header fields */
    hdr = state->hdr_template;
    hdr->Length = msgLength(msg);
    if (state->mid == MAX_MID)
        state->mid = 0;
    hdr->MID = ++state->mid;

    /* attach header to msg and send it */
    buf = msgPush(msg, HDR_LEN);
    chan_hdr_store(hdr, buf, HDR_LEN);
    xPush(xGetDown(self, 0), msg);

    /* schedule first timeout event */
    state->retries = 1;
    state->event   = evSchedule(retransmit, self, state->timeout);

    /* wait for the reply msg */
    semWait(&state->reply_sem);

    /* clean up state and return */
    flush_msg(state->request);
    state->status = IDLE;
    return state->ret_val;
}
```

The next routine (retransmit) is called whenever the retransmit timer fires. It is scheduled for the first time in chanCall, but each time it is called, it reschedules itself. Once the request message has been retransmitted four times, CHAN gives up: it sets the return value to XK_FAILURE and wakes up the blocked client process. Finally, the reason retransmit first checks to see if the event was cancelled is that there is a potential race condition between when evCancel is invoked and when the event actually executes. Note that each time retransmit executes and sends another copy of the request message, it needs to re-save the message in state variable request. This is because each time a protocol calls the xPush operation on a message, it loses its reference to the message.

```
static void
retransmit(Event ev, int *arg)
{
    Sessn     s = (Sessn)arg;
    ChanState *state = (ChanState *)s->state;
    Msg       tmp;

    /* see if event was cancelled */
    if (evIsCancelled(ev))
        return;

    /* unblock the client process if we have retried 4 times */
    if (++state->retries > 4) {
       state->ret_val = XK_FAILURE;
       semSignal(state->rep_sem);
       return;
    }

    /* retransmit request message */
    msgConstructCopy(&tmp, &state->request);
    xPush(xGetDown(s, 0), &tmp);

    /* reschedule event with exponential backoff */
    evDetach(state->event);
    state->timeout = 2*state->timeout;
    state->event = evSchedule(retransmit, s, state->timeout);
}
```

CHAN's chanPop routine is very simple. This is because CHAN is an asymmetric protocol: the code that implements CHAN on the client machine is completely distinct from the code that implements CHAN on the server machine. In fact, any given CHAN session will always be a purely client session or a purely server session, and this fact is stored in a session state variable (type). Thus, all chanPop does is check to see whether it is a server session (one that expects REQ messages), or a client session (one that expects REP messages), and calls the appropriate client- or server-specific routine. In this case, we show only the client-specific routine.

```
static XkReturn
chanPop(Sessn self, Sessn lls, Msg *msg, void *inHdr)
{
    /* see if this is a CLIENT or SERVER session */
    if (self->state->type == SERVER)
        return(chanServerPop(self, lls, msg, inHdr));
    else
        return(chanClientPop(self, lls, msg, inHdr));
}
```

The client-specific pop routine (chanClientPop) is given below. This routine first checks to see if it has received the expected message, for example, that it has the right MID, the right BID, and is of type REP or ACK. This check is made in subroutine clnt_msg_ok (not shown). If it is a valid acknowledgment message, then chanClientPop cancels the retransmit timer and schedules the probe timer. The probe timer is not shown, but would be similar to the retransmit timer given above. If the message is a valid reply, then chanClientPop cancels the retransmit timer, saves a copy of the reply message in state variable reply, and wakes up the blocked client process. It is this client process that actually returns the reply message to the high-level protocol; the process that called chanClientPop simply returns back down the protocol stack.

```
static XkReturn
chanClientPop(Sessn self, Sessn lls, Msg *msg, void *inHdr)
{
    ChanState *state = (ChanState *)self->state;
    ChanHdr   *hdr = (ChanHdr *)inHdr;

    /* verify correctness of msg header */
    if (!clnt_msg_ok(state, hdr))
        return XK_FAILURE;

    /* cancel retransmit timeout event */
    evCancel(state->event);

    /* if this is an ACK, then schedule PROBE timer and exit*/
    if (hdr->Type == ACK) {
       state->event = evSchedule(probe, s, PROBE);
       return XK_SUCCESS;
    }

    /* msg must be a REP, so save it and signal blocked client process */
    msgAssign(state->reply, msg);
    state->ret_val = XK_SUCCESS;
    semSignal(&state->reply_sem);

    return XK_SUCCESS;
}
```

9.4 Purely Synchronous Protocol

We next show a purely sychronous protocol, called SELECT, that is typically configured on top of CHAN. SELECT's job is to dispatch request messages to the appropriate procedure. What this means is that on the client side, SELECT is given a procedure number that the client wants to invoke, puts this number it its header, and invokes a lower level request/reply protocol like CHAN. When this invocation returns, SELECT merely lets the return pass on through to the client; it has no real demultiplexing work to do. On the server side, SELECT uses the procedure number it finds in its header to select the right local procedure to invoke. When this procedure returns, SELECT simply returns to the low-level protocol that just invoked it.

The x-kernel code for selectCall and selectCallPop is given below. Notice that SELECT is like CHAN in that it is asymmetric—each session is either a client session or a server session. Unlike CHAN, however, SELECT exports the synchronous interface to both higher-level protocols (xCall) and lower-level protocols (xCallDemux and xCallPop).

```
static XkReturn
selectCall(Sessn self, Msg *msg, Msg *rmsg)
{
    SelectState *state = (SelectState *)self->state;
    char        *buf;

    buf = msgPush(msg, HLEN);
```

```
    select_hdr_store(state->hdr, buf, HLEN);
    return xCall(xGetDown(self, 0), msg, rmsg);
}

static XkReturn
selectCallPop(Sessn s, Sessn lls, Msg *msg, Msg *rmsg, void *inHdr)
{
    return xCallDemux(xGetUp(s), s, msg, rmsg);
}
```

9.5 Virtual Protocols

In the x-kernel, we sometimes implement modules called *virtual protocols*. Virtual protocols are configured into a protocol graph just like any other protocol, but they are different from regular protocols in that they do not add a header to messages; i.e., they do not directly communicate with their peer on the other machine. Instead, virtual protocols serve to "route messages" through the protocol graph. For example, the virtual protocol VSIZE is configured on top of some number of low-level protocols (usually two). VSIZE takes a message from some high-level protocol, and based on how big the message is (this can be determined using the x-kernel's msgLength operation), decides which of the two low-level protocols to pass the message on to. For example, VSIZE might be configured into the protocol graph to route large messages (those greater than 1KB) to BLAST, and small messages (those less than or equal to 1KB) to IP. The advantage of using a protocol like VSIZE is that it allows messages that are too small to need fragmenting from incurring the overhead of yet another protocol layer.

The following shows VSIZE's open routine. The main thing to notice about vsizeOpen is that it passes the hlpType it was given from above as the hlpType to the protocol below it, rather than using self. This is because VSIZE wants to intercept all of the higher-level protocol's messages, and it has no header of its own to stick this protocol's demux key in.

```
static Sessn
vsizeOpen(Protl self, Protl hlp, Protl hlpType, Part *p)
{
    Sessn  s;
    Sessn  lls[VSIZEMAXDOWN];
    Part   savedPart[3];
    PSTATE *pstate = (PSTATE *)self->state;
    int    plen;
    int    i,j;

    /*
     * Save the original participants before opening since we need to
     * use it in both opens and it will get munged in the first open
     */
    plen = partLength(p) * sizeof(Part);
    bcopy((char *)p, (char *)savedPart, plen);

    for (i = 0; i < pstate->numdown; i++) {
        lls[i] = xOpen(self, hlpType, xGetProtlDown(self, i), p);
        if (lls[i] == ERR_SESSN) {
            /* could not open session */
            for (j = 0; j < i; j++)
                xClose(lls[j]);
            return ERR_SESSN;
```

```
        }
        bcopy((char *)savedPart, (char *)p, plen);
    }
    if (mapResolve(pstate->activeMap, lls, (void **)&s) == XK_SUCCESS) {
        /* found an existing session */
        for (i = 0; i < pstate->numdown; i++)
            xClose(lls[i]);
    }
    else {
        /* creating a new session */
        if ((s = vsizeCreateSessn(self, hlp, hlpType, lls)) == ERR_SESSN) {
            for (i = 0; i < pstate->numdown; i++)
                xClose(lls[i]);
        }
    }
    return s;
}
```

VSIZE's push routine is very simple—it checks the size of the message pushed to it, and calls the appropriate session below it. The code for vsizePush follows.

```
static XkHandle
vsizePush(Sessn s, Msg *msg)
{
    SSTATE *sstate;
    int    i;

    sstate = (SSTATE *)s->state;
    for (i = 0; i < sstate->numdown-1; i++) {
        if (msgLength(msg) <= sstate->cutoff[i])
            return xPush(xGetSessnDown(s, i), msg);
    }
    return xPush(xGetSessnDown(s, sstate->numdown-1), msg);
}
```

References

[1] D. Mosberger. *Map Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[2] D. Mosberger. *Message Library Design Notes*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[3] Network Systems Research Group, Department of Computer Science, University of Arizona. *x-kernel Programmer's Manual (Version 3.3)*, Jan. 1996.

[4] L. L. Peterson. *Getting Started with the x-kernel*. Network Systems Research Group, Department of Computer Science, University of Arizona, Jan. 1996.

[5] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, San Fransico, CA, 1996.

[6] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 25–38, Nov. 1987.