## TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# AN IMPLEMENTATION MODEL FOR CONNECTION-ORIENTED INTERNET PROTOCOLS

## 1. INTRODUCTION

Recently, a number of research groups have proposed connection-oriented access protocols at the network and internet layer. These protocols share four important characteristics. First, the path that data packets take from source to destination is established in advance. Second, the resources required for a connection are reserved in advance. Third, the connection's resource reservation is enforced throughout the life of the connection. Finally, when the connection's transaction is finished, the connection is broken and the allocated resources are freed.

In a high-speed communication environment, these connection-oriented access protocols have some advantages over traditional connectionless service. First, the internet should be able to support a variety of applications which require transport of voice, video, and data. Typical target applications include video distribution, computer imaging, distributed scientific computing and visualization, distributed file access, and multimedia conferencing. These applications generate traffic with very different requirements in terms of expected throughput, delays, errors, and their ability to dynamically adjust resource requirements. In order to support and efficiently handle the quality-of-service requirements of such a varied set of applications over an internet, it is important to be

able to use the application's traffic characteristics to make a decisions about routing and to make statistical guarantees to the application. Indeed, if connectionless packet delivery protocols were used, it would not be possible to make use of the application's behavior to help give it any assurances, or even to improve the efficiency or utilization of the networks. With the aid of connection-oriented access protocol's resource reservations made on the basis of the application's traffic patterns, it would become possible to achieve both of these goals [11].

Second, most of the emerging high speed networks, such as ATM [5], and PLANET [2, 3], recognize the needs of new applications, and provide connection-oriented access. Connection-oriented protocols also simplify per-packet processing due to the state information stored along the path, and hence make it more appropriate for high speed operation.

It is also important to note that the communication environment is and will continue to be an internet of many heterogeneous networks. Clearly, future internets will include emerging high speed networks as components, and applications will continue to treat the internet as the virtual network.

Thus, it is important that the internet-level protocol be able to provide high bandwidth and performance guarantees to applications. This has prompted a number of research groups, including our group at Washington University in St. Louis, to propose connection-oriented internet protocols (COIPs), which include MCHIP [9], ST [6, 12], and FLOW [13, 14]. The Internet Activities Board (IAB) and Internet Engineering Task Force (IETF) also recognized the need for exploring connection-oriented internetworking and initiated a COIP working group.

## 1.1. MOTIVATION

The proposed COIPs have a number of similarities and differences. The members of the COIP working group decided that it is important to pursue these protocols and compare and contrast the alternate approaches for implementing them. However, implementation of these protocols completely independently was considered unwise for the following reasons. First, as the proposed COIP protocols have a number of common functions, independent implementations would lead to a lot of duplicate work. For example, all the protocols have a connection state machine and a resource allocation and enforcement function. Secondly, implementation of a protocol in the Unix kernel poses a number of challenges: coding or logic errors can result in system crashes, kernel debugging support is limited, kernel dynamic memory allocation mechanisms are complex, the protocol's code must co-exist with the rest of the kernel, and the existing kernel interface is not well documented.

In order to develop a more productive research environment, avoid duplication of work, and foster collaboration, we proposed the COIP-kernel (COIP-K). COIP-K forms the core of a COIP protocol and includes the minimum functionality necessary for a wide range of multicast connection-oriented protocols. It also includes appropriate provisions to interface with other functional modules. COIP-K, when combined with a set of functional modules, will create an instance of a COIP such as MCHIP or ST. This process is shown in Figure 1.1.

This approach to protocol development yields important benefits to the research effort. Many of the functions that a COIP must provide can be supported by a number of alternative mechanisms. These mechanisms can be implemented in different experimental modules and integrated with COIP-K to produce different instantiations of COIPs.

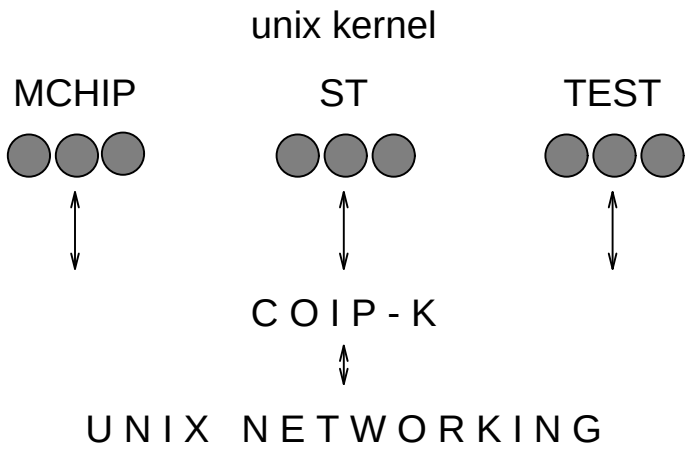



Figure 1.1: COIP-K structure

These instantiations represent different mechanisms which can be compared under controlled experimental conditions. As a result, it will be possible to describe under what conditions each of the alternate mechanisms behave well or poorly, and thus define a COIP that is optimal for a given target environment.

Additionally, by providing a set of default functional modules, COIP-K can provide an incremental level of support to a protocol programmer. For example, a novice COIP-K user could use mostly default COIP-K functional modules and get a simple test protocol up and running quickly. As the novice COIP-K user got more advanced, he or she could swap out more and more default modules in favor of of his or her own modules to make a more sophisticated protocol. By providing incremental support, COIP-K can simultaneously provide strong support for novice users and less, non-obtrusive support for advanced users.

COIP-K includes routing and resource functions to set up a connection, functions to forward data packets based on connection identifiers, and functions to terminate the connection. Thus, COIP-K can run the basic state machine necessary for a connection-oriented protocol, and its implementation in the Unix kernel can provide the standard

interface to its higher-level protocols. However, it is important to note that COIP-K leaves a number of options open and delegates important decision making to other functional modules. For example, COIP-K talks to the resource manager for resource availability and allocation, but the actual resource allocation algorithm is part of the resource allocation module.

COIP-K, by default, provides only a very basic connection management scheme. It assumes that all endpoints of a connection are known at connect time, and that they can not be added or deleted after a connection is established. Also, COIP-K's default concept of connection establishment is not very reliable (it depends on a two-way handshake and timers). To provide more elaborate connection management, more complex protocol-specific functional modules must be provided.

The topic of this thesis is the challenge of realizing the COIP-K vision and demonstrating its feasibility and viability. The goals of this thesis can be realized by meeting a set of implementation requirements as explained in the next section.

## 1.2. COIP-K IMPLEMENTATION REQUIREMENTS

COIP-K has five main implementation requirements which define the scope of this research. The requirements are the following:

- COIP-K must be implemented in the Unix kernel. Protocols implemented outside the kernel (in a user process) are inefficient due to context switching and the cost of multiplexing data from one user process to another.

- COIP-K should allow implementation of various COIP protocols by module interchange. COIP-K module interchange should also provide incremental software support by providing a set of overrideable default modules. This allows COIP-K

users to easily build protocols and to easily compare and contrast the different tradeoffs associated with the protocols.

- COIP-K should refrain (as much as possible) from modifying the user-level socket interface. This will allow easy porting of applications from traditional protocols, to COIP for testing.

- COIP-K should have efficient per-packet processing. Eventually, once the tradeoffs associated with different COIPs are explored, the critical path of a COIP protocol should be implemented in hardware. Having simple and efficient per-packet processing will facilitate this.

- Since most COIP protocols support multipoint connections, COIP-K must support them too.

## 1.3. THESIS OUTLINE

Chapter 2 presents background information and necessary details of the 4.3 BSD Unix operating system's networking system. This networking system is divided into three software layers whose data structures and functions are presented in turn. Memory management (mbufs) and the calling sequences between layers are also presented.

Chapter 3 presents the details and design choices of the COIP-K implementation in the context of the background information of Chapter 2. The application programmer interface, various data structures, and functions that constitute COIP-K in the protocol layer are presented. Chapter 3 also presents the different COIP-K modules that serve as the toolbox for other COIP protocols.

Chapter 4 discusses the feasibility and viability of COIP-K. The specification and implementation of the COIP Test Protocol (CTP) is first presented. This demonstrates

COIP-K's usefulness in creating implementations of COIP protocols. Then, the concept of module interchange, which is critical in realizing different COIPs using COIP-K, is presented. Next COIP-K's performance is presented to characterize the per-packet processing and to quantify the cost of using the COIP-K concept. COIP-K is shown to be efficient, having delay and throughput close to that of UDP. Finally, a number of COIP-K demonstration programs are presented.

Finally, Chapter 5 summarizes the contributions made by this research and discusses directions for future research.

# 2. NETWORKING BACKGROUND

## 2.1. INTRODUCTION

This chapter presents background information on the 4.3 BSD Unix networking system, using the Internet protocols as examples. It is based in part on information found in [8] and detailed study of the SUNOS source code. This background is important for this thesis for two main reasons:

- A developer wanting to use COIP-K to build a COIP protocol would have to have a firm grasp of the networking background information, which is not easy to acquire. Thus, by presenting the results of this study here we hope to save other COIP developers time.
- The knowledge of the networking environments also helps one to fully appreciate COIP-K's complexity and tradeoffs.

Networking in the BSD Unix kernel is divided into three software layers: the socket layer, the protocol layer, and the network interface layer. These layers are shown in Figure 2.1.

The socket layer is the top layer from the user's point of view. This layer provides a generic, uniform, protocol-independent interface to networking services for the applications programmer. At the programming level, the socket layer appears to consist of a standard set of C functions which handle all network interaction. These functions are actually system calls which cause the system to enter kernel mode and call down to the lower networking layers.

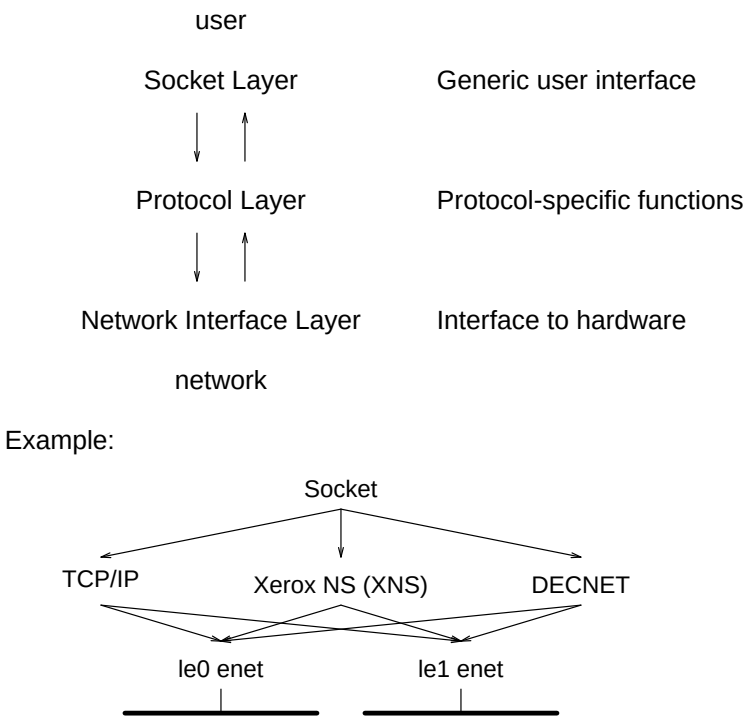



Figure 2.1: BSD Unix network layering

The middle layer, called the protocol layer, consists of a number of protocol suites or protocol domains. The protocol layer handles all protocol-specific processing of network data and includes implementations of various protocols such as TCP, IP, and XNS.

The third and lowest layer in the BSD networking environment is the network interface layer. The software in this layer consists of network device drivers which provide an interface to the computer's networking hardware.

Figure 2.1 shows how the three layers are organized with respect to the user and the network. In the example at the bottom of Figure 2.1, the socket layer can choose between three domains (sets of related protocols). Each protocol in a domain can route to different network interfaces. Thus, a user can choose a protocol from one of the TCP/IP, XEROX NS, or DECNET domains. All the protocols can co-exist on the same network. The next three sections describe each layer in more detail. The last two

sections present the details of the memory allocation system (`mbufs`), and the calling sequences between the three layers, respectively.

## 2.2. THE SOCKET LAYER: CLIENT-SERVER MODEL

The socket layer is the means by which a user interfaces to the networking system. It consists of a set of system calls and some protocol-specific data structures. These system calls are protocol independent, thus the application programmer interface between different protocols (e.g. a COIP-K protocol and TCP) is very similar. The socket layer's generic interface supports a client-server model as a paradigm for communications. The client-server model is used for interprocess communications when two processes are involved. One process, called the server, is usually running in the background waiting to service requests. When another process starts up and connects to the server, it is called a client. The client and server processes can be on the same machine or on machines across an internet. The next two subsections take a closer look at clients and servers.

### 2.2.1. CLIENT SIDE

The left side of Figure 2.2 shows typical socket system calls used to establish and use a connection on the client side. In general, these calls return $-1$ if there is an error and set the global variable `errno` to the value of the error that occurred. The `socket()` system call causes the kernel to create a socket structure and associate it with the process making the `socket()` call. The system call returns a file descriptor. This file descriptor, once set up, can be used in normal I/O operations, just like a file or pipe. Arguments to the socket system call specify which protocol is attached to the socket. The `family` parameter selects which protocol family (or protocol suite/domain) to use. For example, `PF_INET` would select the internet protocol family which includes TCP, UDP, and raw IP. The `type` field is used to select the type of socket desired. Some valid

```
        CLIENT                                  SERVER

s = socket(family, type, protocol)      s = socket(family, type, protocol)

err = connect(s, addr, addrlen)         err = bind(s, addr, addrlen)

                                        err = listen(s, 5)

                                        s2 = accept(s, addr, addrlen)


err = read(s, buf, buflen)              err = read(s2, buf, buflen)

err = write(s, buf, buflen)             err = write(s2, buf, buflen)

close(s)                                close(s2))
```

Figure 2.2: A typical client (left) and server (right)

types for a socket are stream, datagram, and raw. The `protocol` field specifies which protocol of type `type` should be used. Generally there is only one protocol of each type, so this parameter is usually set to zero. For example, to create a TCP socket, one would select the internet protocol family `INET`, a socket type of stream `SOCK_STREAM`, and a protocol number of 0. To create a UDP socket, the type should be changed to `SOCK_DGRAM`.

Once a socket has been created, it can be connected to a remote server (a client) or it can wait for connections (a server). Connecting to a remote server is done with the `connect()` system call, as in Figure 2.2. The parameter `s` is the socket file descriptor to use in the connection attempt. The parameters `addr` and `addrlen` are used to pass the protocol-specific addressing information about the remote host to the system. On 4.3 BSD systems the length of the `addr` buffer is limited to 112 bytes, but this is easy to extend to 1024 bytes (as has been done in SUNOS). The variable length of the addressing parameters allow some flexibility in what sort of protocols can be implemented in BSD Unix[1]. In particular, it allows protocols which support multipoint connections to be implemented. Under `INET` protocols, the addressing information is stored in a structure called `sockaddr_in`. This structure consists of a IP host number and an IP port number.

[1] although it is clear that arbitrary limits such as the 1024 byte should be removed

Once the `connect()` call is complete, the user can do normal I/O on the socket file descriptor with `read()` and `write()` system calls as shown in Figure 2.2. The buffer `buf` is not limited in length (unlike the address in the `connect()` system call). The socket layer also handles features such as non-blocking I/O and synchronous I/O multiplexing (e.g. the `select()` system call).

Finally, when a connection is no longer needed, either the client or server can close the socket by calling the `close()` system call on it.

### 2.2.2. SERVER SIDE

Creating a socket to accept connections is a bit more difficult. A typical server code is shown in Figure 2.2. Creating a server-side socket with the `socket()` system call is the same as creating a client-side socket. Also, once a connection is established in the server, I/O and shutdown are the same as in the client. However, a server must be able to wait for connections, and this requires additional system calls.

Many protocol families support the notion of services. In the `INET` domain this means that a well-known service program such as the mailer daemon waits on a well-known port number (TCP port 25) for connections. In order to associate a socket with a well-known port number, the `bind()` system call is used. The addressing parameters `addr` and `addrlen` specify what local address and port the socket should be associated with. The parameters are limited in length in the same way as they were in the `connect()` system call. Using `bind()` is optional on most protocols. If an application program does not bind to a specific address or port number, then the system randomly assigns one that is not in use by other processes or well-known servers (the server can find out its local port number with the `getsockname()` system call).

After binding, the socket is ready to be marked as a listening socket. This is done with the `listen()` system call. The `listen()` call takes two parameters: the socket file descriptor which is to be marked as listening, and the maximum number of unprocessed connection requests the system will queue before dropping them (generally limited to five). Note that `listen()` only marks a socket as capable of receiving connections — it does not actually accept connections.

Once the socket is listening for connections, it can use the `accept()` system call to accept a connection. The `accept()` function call returns a brand new socket which is connected to the remote end so that the original socket can be used to listen for subsequent connection requests. If there are no connections pending, `accept` puts the user's process to sleep until a connection request is received. If the addressing parameter `addr` is non-null then the remote machine's address in written into the `addr` buffer after the connection is made.

Once a connection has been accepted, most server programs fork a copy of the server process to service the new connection (on `s_new`) and continue to wait for new connection requests.

## 2.3. THE PROTOCOL LAYER

The protocol layer consists of a number of domains, such as the Internet domain (`INET`), the XEROX NS domain, and the DECNET domain. Each domain includes a number of protocols, as shown in Figure 2.3. For example, the `INET` domain includes the TCP, UDP, and raw IP protocols. COIP-K is an addition to this layer; therefore to understand COIP-K internals, it is important to understand the protocol layer.

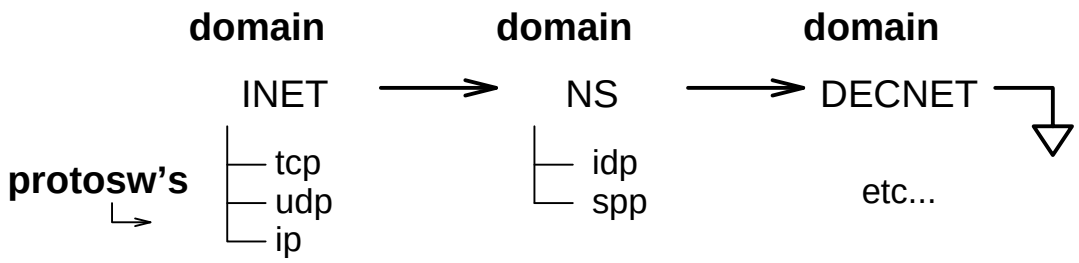



Figure 2.3: Protocol layer setup

### 2.3.1. DATA STRUCTURES

The protocol layer uses three main types of data structures: the `domain` structure, the `protosw` or protocol switch structure, and a protocol-specific structure called a protocol control block (PCB). Each of these are explained in the following subsections.

**DOMAIN DATA STRUCTURE.** The `domain` structure contains pointers to all information concerning a specific protocol family. The host system creates a linked list of `domain` structures at the time of start up. Each domain has a corresponding well-known identification number which is used by applications to specify the domain to be used (the first argument in a `socket()` call). The protocol `domain` structure contains:

- a unique ID number (e.g. `PF_INET` from `socket.h`)
- an ASCII name
- a pointer to an initialization function
- access rights (used in the Unix domain)
- a list of protocols supported by the domain (e.g. TCP, UDP, raw IP)

**THE PROTOCOL SWITCH DATA STRUCTURE.** The list of protocols supported by a domain is actually a list of protocol switch (`protosw`) structures. These

structures totally define an interface to a specific protocol under a domain. A `protosw` structure consists of some data and a set of pointers to functions. The data in the `protosw` structure consists of:

- socket type and protocol (as in the `socket()` system call)
- a back pointer to the domain structure, of which the protocol is a member
- flags to indicate some basic properties of the protocol (e.g. if a connection is required before data is sent)

The functions that the `protosw` structure points to are described in Section 2.3.2.

**PROTOCOL CONTROL BLOCK DATA STRUCTURE.** A PCB, or protocol control block, is where all protocol state information on a socket is stored. Each socket has its own PCB (thus, there is one PCB per end of a connection). PCBs are typically stored as a linked list of memory buffers in the kernel. Since PCBs are private to the protocol layer, their structure is not strictly defined by the protocol layer. Some domains, such as `INET`, define both a main PCB that is common to all protocols in the domain and an optional per-protocol control block that is used for more protocol specific information, as shown in Figure 2.4. In this case the main PCB stores information such as source and destination addresses, port numbers, routing information, and IP options. This is enough information for UDP; a per-protocol PCB is not needed. However, TCP requires a per-protocol PCB to store additional state information such as timers, flags, packet templates, and TCP state.

### 2.3.2. PROTOCOL LAYER FUNCTIONS

The protocol switch structure includes pointers to several standard functions which define the operating system's interface to a specific protocol. These functions are:

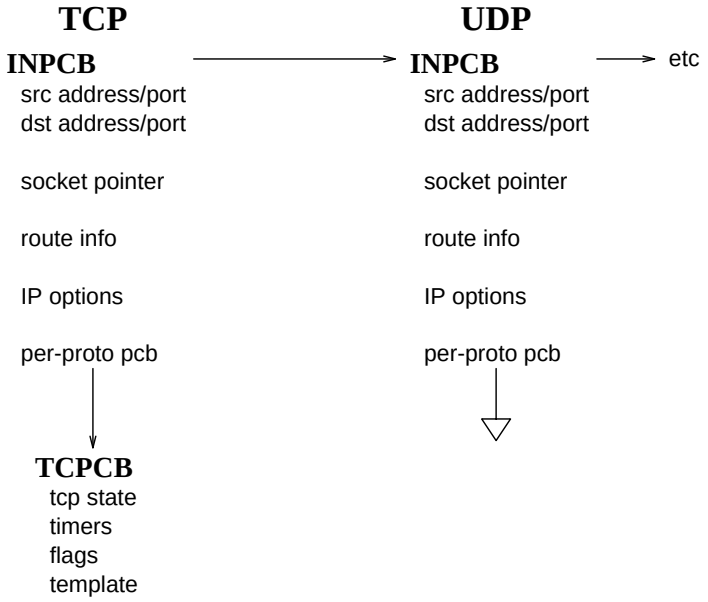



Figure 2.4: Internet domain PCB structure

**User request function:** This function is the interface between the socket layer and the protocol layer. Any protocol that interfaces with the socket layer must have a user request function; all communications from the socket layer to the protocol layer go through this function. The user request function has one integer parameter called the request number. This number tells the protocol what action it is expected to perform. For example, when a user program performs a `write()` on a socket file descriptor, the protocol issues a `PRU_SEND` request. When a user program creates a socket, the protocol issues a `PRU_ATTACH` request.

**Input/Output functions:** There are four input/output functions. One set of I/O functions is for passing data between stacked protocols (e.g. TCP and IP), or for passing data from a protocol to another layer (e.g. from IP_OUTPUT to the network interface layer). The other set of I/O functions is for passing control information between protocols.

**Init function:** This function is called only once, at system startup time. This function must create and initialize any private data structures which a protocol uses. For example, a protocol may keep a linked list of active connections, and it may need to make that list null to start with.

**Timer functions:** These functions, if present, are called on a regular interval by the operating system. The slow timer is called every 500 ms, and the fast timer is called every 200 ms. These timers can be used to do tasks such as retransmission and connection time out.

**Drain function:** When the system is running low on buffer space, this function is called to ask a protocol to free up as much memory as possible.

### 2.3.3. ROUTING

The protocol layer must choose which interface to output its packets on by using a routing function. IP routing is done using the `rtalloc()` function which takes an IP address and returns a structure. This structure contains the information necessary for the protocol layer to determine which network interface data structure to use when outputting data.

## 2.4. THE NETWORK INTERFACE LAYER

The network interface layer controls the actual network hardware on the machine. It exchanges data in the form of packets with the protocol layer. Each network interface has a network interface structure associated with it. This structure is called `ifnet`, and it consists of:

- a name which indicates what type of device the interface is (e.g. "le" on the SUNOS system refers to the Lance Ethernet chip)

- a unit number
- a link to the next interface (interfaces are stored in a linked list)
- the maximum transmission unit (MTU) and flags
- a function which outputs a packet on a network device
- a function which handles device configuration requests (`ioctl` function)
- a function to initialize and reset the network device hardware
- various statistic information
- a linked list of addresses for the interface which consist of:
  - local address
  - broadcast address and point to point address
  - pointer to next address
  - back pointer to `ifnet` structure

For example, Figure 2.5 shows a Sun system with two ethernet interfaces `ie0` and `ie1`. The `ie0` ethernet supports both IP and DECNET protocols, thus the `ifnet` structure has two addresses (i.e. the local IP address and the local DECNET address). Likewise, the `ie1` interface supports IP and XEROX NS protocols and has those two local addresses.

The protocol layer communicates to the network interface layer via the `ifnet` output function. The output function receives a packet and either sends it or queues it for output. The network interface layer communicates with the protocol layer by calling the protocol layer interrupt function when a packet for that layer is received on the network.

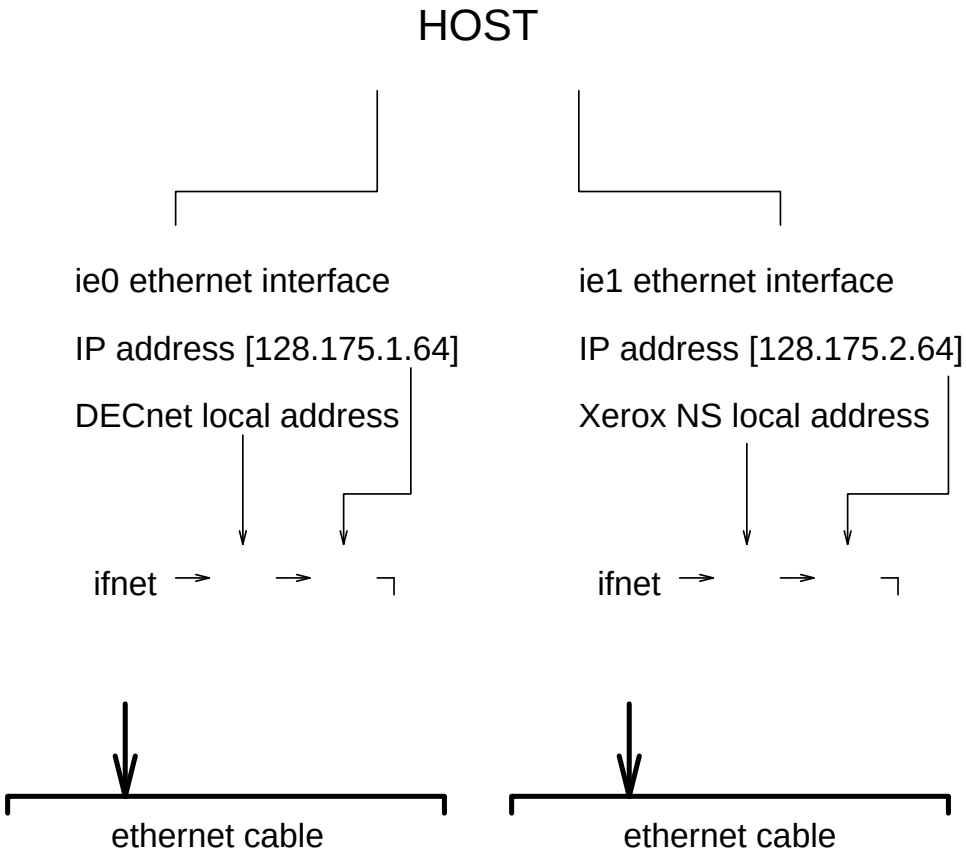



Figure 2.5: An example of network interface structure

## 2.5. MEMORY BUFFERS

Another important aspect of the 4.3 BSD Unix networking system is the dynamic memory allocation system. This system is called the **mbuf** system (mbuf stands for memory buffer). A mbuf is a 128 byte data structure used in storing data such as user data buffers, packets, and protocol control blocks. Data can be stored in a mbuf in one of two ways depending on what type of mbuf is used. In a **small mbuf**, the data is stored within the 128 byte mbuf structure. Of the 128 bytes in the mbuf, 112 are used for data storage and the other bytes are used as pointers to maintain a chain of mbufs, as explained later. On the other hand, in a **large-mbuf**, the 128 mbuf structure contains

a pointer to a physical page of memory where the data is stored. Typically the size of a page of memory is 1024 bytes. The mbuf system maintains its own private pool of pages for large mbuf allocation.

The mbuf dynamic memory allocation system was designed to meet two main objectives. The first objective is that it must be able to support encapsulation and decapsulation without the copying of data. The second objective is that dynamic memory allocation must not fragment physical memory.

### 2.5.1. MBUF CHAINS

Because data generally does not come in even units of 112 or 1024 bytes, separate mbufs are linked together into a linked list to store a piece of data of any size. This is done with the `m_next` pointer in the mbuf structure. Mbufs linked together with the `m_next` pointer are called **mbuf chains**. Also, since mbufs are used to store data such as packets, it is generally useful to be able to link mbuf chains into a linked list to create a packet queue. The `m_act` pointer in the mbuf structure can be used to do this.

An example of the use of the two types of mbuf pointers is shown in Figure 2.6. Each of the four blocks is one mbuf. The two mbufs on the top and the two mbufs on the bottom are linked together with the `m_next` field to form two mbuf chains. In this example, each chain is storing a single packet. The packets in the two mbufs chains are linked together with the `m_act` pointer to form packet queues. The "`MT_`" fields in the figure specify the "type of data" stored in the mbufs.

### 2.5.2. MBUF MANIPULATION

While a single mbuf can store up to 112 or 1024 bytes (depending on the type of mbuf), not all of those bytes are valid data. The mbuf has two fields which determine how much of the data is actually valid. The `m_off` field indicates the offset of the first byte

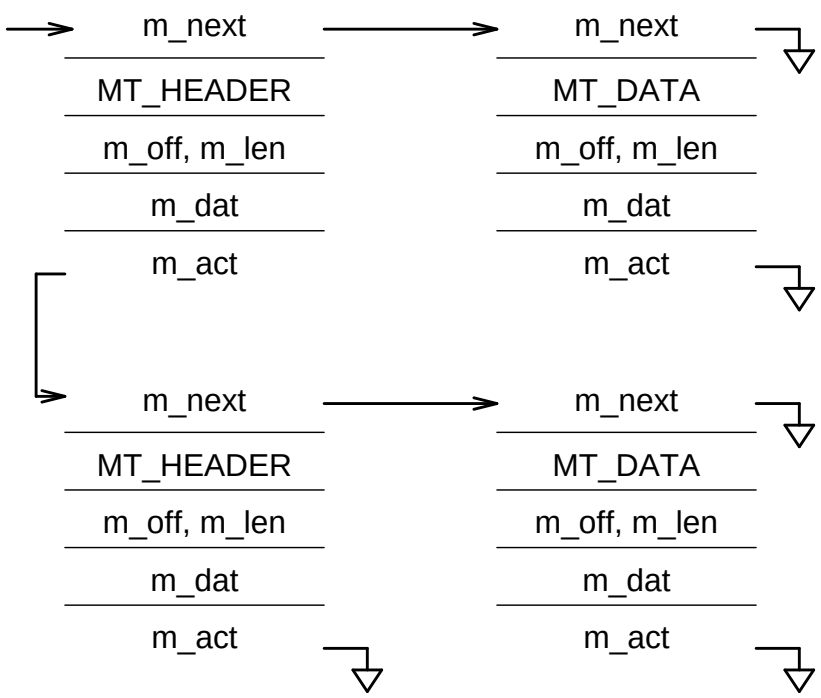



Figure 2.6: Dynamic memory allocation with mbufs

of the data in the mbuf. The `m_len` field indicates how many bytes of valid data are actually in the mbuf. Together these two fields are useful for protocol processing of packets in mbufs, especially for layers of protocols. As data moves between different network layers it is encapsulated or decapsulated in the appropriate packet formats. This involves addition or removal of header information to the data. With the `m_off` and `m_len`, fields it is easy to achieve packet encapsulation and decapsulation without having to copy the data over and over again. For example, Figure 2.7 shows how a header can be removed from an mbuf containing a packet by simply adding the length of the header to the `m_off` field, and subtracting the same amount from the `m_len` field. By using the length and offset fields, or by appending (or prepending) new mbufs to a chain, it is possible to avoid memory to memory copies. Also, the fixed size of an mbuf reduces memory fragmentation.

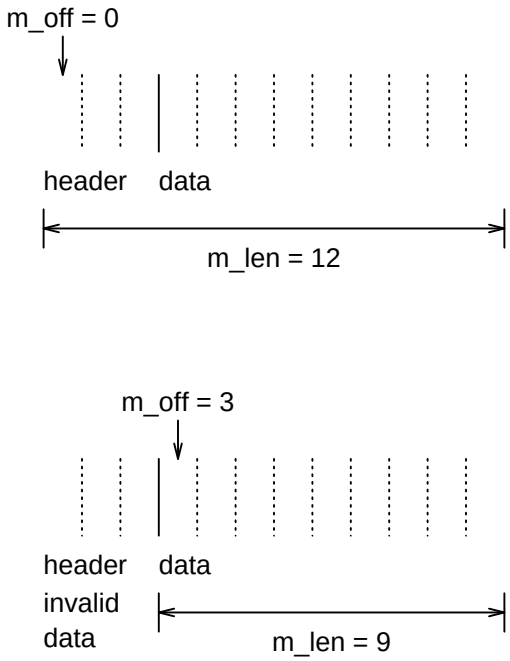



Figure 2.7: Removing data from an mbuf

### 2.5.3. MBUF PROGRAMMER INTERFACE

A programmer's interface to the mbuf system consists of a few functions and macros. The `m_get()` function call is used to get an mbuf. This function call takes two arguments and returns a pointer to a small mbuf (or NULL if there is an error). The function to get a large mbuf is `mclget()`. This function takes a small mbuf as an argument and converts it to a large mbuf. It returns `NULL` if it can not get a large mbuf.

Once an mbuf has been allocated the programmer can adjust the `m_len`, `m_act`, `m_next`, and `m_act` fields as needed.

The `mtod()` macro is used to access data in an mbuf. This macro takes a pointer to an mbuf and converts it to a pointer to the data in the mbuf as specified in the second argument. For example, if an mbuf contained entire packet and there was a `struct header` which defined the packet header format, the top C program segment in Figure 2.8 can be used to obtain a pointer to the data. After this sequence of calls, the pointer

`hp` can be used like a normal pointer to a structure. There is also a C macro `dtom()` which takes a pointer to data in an mbuf and changes it into a pointer to that mbuf.

**The mtod macro:**

```
struct header *hp;
struct mbuf *mb;

hp = mtod(mb, struct header *);
```

---

**The m_pullup function:**

```
if (m->m_len > sizeof(struct header))
    m = m_pullup(m, sizeof(struct header));
```

Figure 2.8: Programming interface to mbufs

Often a programmer wants to treat the data in an mbuf like a structure, as shown above. However, the mbuf system does not ensure that the entire header will be in the same mbuf. For example, the header of a packet stored in an mbuf chain could be in both the first and second mbufs in the chain. To solve this problem there is a function called `m_pullup()` which takes a pointer to an mbuf chain and a length `L`, and ensures that that the first `L` bytes of data in the chain are stored in a contiguous area of memory (in the first mbuf). The usage of this function call is shown in the bottom of Figure 2.8. If the pullup function returns NULL then it failed and pullup function has freed the mbuf chain. Otherwise, it is safe to assume that there are at least `sizeof(struct header)` bytes in the first mbuf of the chain.

Once the program is done with an mbuf, it must free it. There are two functions which free mbufs: `m_free()` and `m_freem()`. The first frees a single mbuf; the second frees an entire mbuf chain.

### 2.5.4. MBUF DATA MOVEMENT

Another important point to note about mbufs is how data is transferred to and from them. First consider data from a user-level `write`. The data first encounters the socket layer. The socket layer gets a pointer to a buffer in user memory and a length. It then copies the data from the user's memory space to kernel space (with the `uiomove()` function). This copy goes directly into an mbuf, so that when the data is presented to the protocol layer, it is already in the mbuf format. The protocol layer adds necessary headers to the mbuf chain with data in it and then passes the chain to the network interface layer. The network interface layer then passes data from the mbuf to the network in a hardware-dependent way[2].

In the reverse direction, data from the network is received using an interrupt from a specific piece of hardware. The network interface layer must receive this data from the network layer, copy it in an mbuf, and then put the mbuf in a packet queue for the protocol layer to process. The protocol layer can then process the mbuf, removing headers and extracting data (via pointer manipulation), until it is ready to pass the mbuf to the socket layer with `sbappend()`. The socket layer receives the mbuf containing the data, copies the data from kernel space to the user's buffer, and then disposes of the mbuf.

An interesting point to note is that the protocol layer deals entirely with mbufs; it never encounters data in any other format.

## 2.6. CALLING SEQUENCES BETWEEN LAYERS

This section presents the details of how the different networking layers communicate among themselves.

[2]This process is beyond the scope of this document.

### 2.6.1. SOCKET - PROTOCOL LAYER INTERFACE

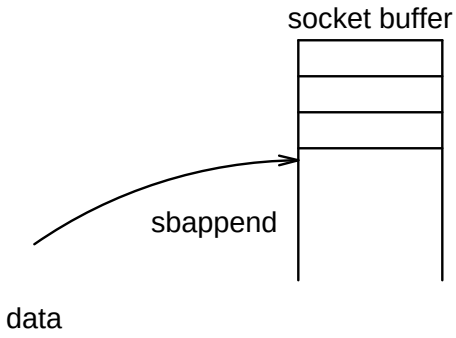
Communications from the socket layer to the protocol layer is done via the user request function of the corresponding protocol (the user request function is discussed in section 2.3.2).

Communications from the protocol layer to the socket layer is a bit more complex. The socket layer keeps track of the state of a connection with its own internal state variable. The protocol layer can change the state of the socket through several functions such as `soisconnecting()`, `soisconnected()`, and `soisdisconnected()`. Non-blocking I/O (I/O in which a process never goes to sleep in a system call) is handled entirely by the socket layer and is transparent to the protocol layer.

Data is passed up to the socket layer through the `sbappend()` function. This function takes a memory buffer and appends it to a socket's inbound data buffer. The socket also provides two functions to wake up any processes which are blocked waiting for data I/O. These functions are `sorwakeup()` and `sowwakeup()` which wake up readers and writers respectively. An example of a typical data transfer from the protocol layer to the socket layer is shown in Figure 2.9. The data buffer `data` is appended to the socket buffer, and any processes that were blocked while reading data are woken up using `sorwakeup()`.

### 2.6.2. PROTOCOL - PROTOCOL COMMUNICATION

Protocols in the protocol layer can be stacked on top of each other. Figure 2.10 shows the TCP/IP implementation layering. Both protocols have their own `protosw` structures. Since a user never deals directly with IP, the IP protocol has no user request function. Instead, all requests to the TCP/IP from a user process must go via the TCP user request function. For example, a write operation on a TCP socket starts at the TCP user request




```
sbappend(&s->so_rcv, data)

sorwakeup(s)
```

Figure 2.9: Data transfer from protocol layer to socket layer

function. The user request function then calls the TCP output function which makes a TCP packet, and then calls the IP output function with the TCP packet as data. The IP output function encapsulates the TCP packet in an IP packet, routes the packet, and calls the network interface layer to send the packet. The network interface layer calls the IP interrupt function `ipintr` when a packet is received from the network. This interrupt function handles the IP input tasks, determines the packet is a TCP packet, and calls the TCP input function. The TCP input function handles the sliding-window protocol and error detection and recovery. The user can then do a read operation which calls down to the TCP user request function and actually gets the data. Other operations such as socket creation and connection establishment are processed via the TCP user request function, as shown in Figure 2.10.

### 2.6.3. PROTOCOL - NETWORK INTERFACE COMMUNICATION

All protocol to network communication is done via the network interface output function. Each network interface has an `ifnet` structure which contains all state information

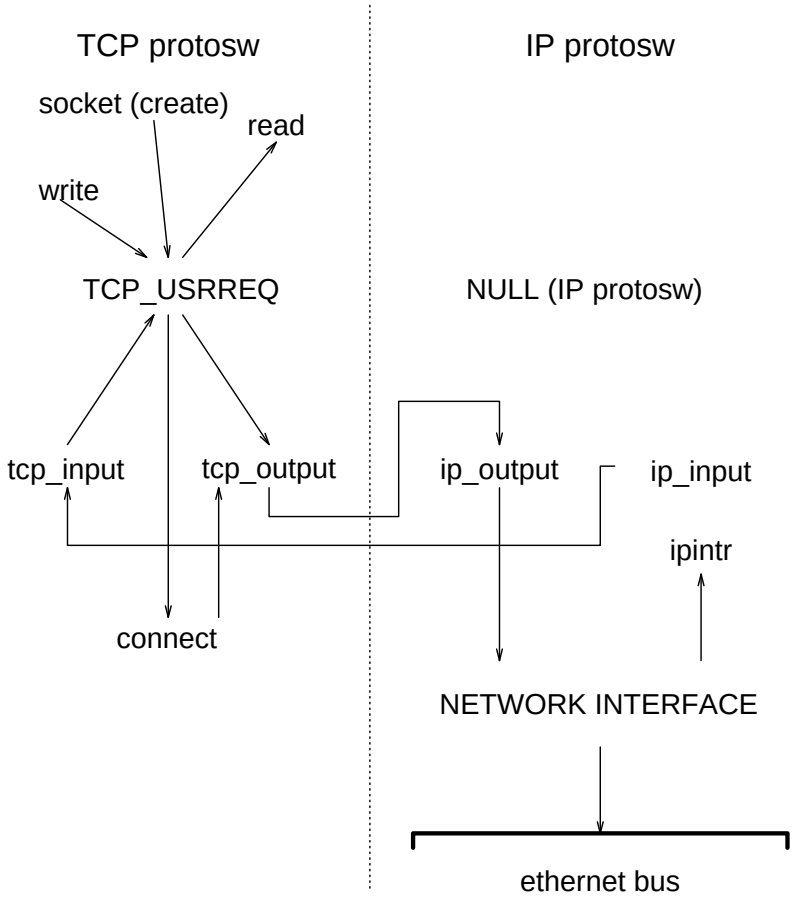



Figure 2.10: Example of how protocols can stack on each other

on that device, including the network interface output function. When the protocol layer decides what route to use to reach a host, it is choosing one of these `ifnet` structures.

Network layer-to-protocol layer communication is done via interrupts. Network devices such as ethernet cards have a high priority interrupt level, so they must be serviced very quickly in order to keep the system from losing interrupts from other devices. The protocol processing, on the other hand, is not as time dependent. So, when the network interface layer receives a packet from the network, it determines which protocol should get the packet, enqueues it on that protocol's queue, and schedules a lower priority interrupt for the corresponding protocol. The lower priority interrupt will call the protocol's interrupt routine and the protocol will then process the packet.

# 3. COIP-K IMPLEMENTATION

## 3.1. INTRODUCTION

This chapter describes the implementation of COIP-K within the framework of the Unix networking model. The application programmer interface for clients and servers is presented first. Then the data structures and functions in the protocol layer are presented. Finally, the COIP-K modules are presented.

## 3.2. APPLICATION PROGRAMMER INTERFACE

The COIP-K application programmer interface uses the standard socket interface to facilitate porting of old applications and development of new applications. No changes to the standard socket layer interface were necessary in our implementation of COIP-K with the exception of the addition of a few new well-known constants for the COIP-K domain. COIP-K uses the client-server model of the standard socket layer for interprocess communication. We shall examine first the socket level data structures that have changed with COIP-K, and then the sequence of calls used by an example client and server.

### 3.2.1. DATA STRUCTURES

The application programmer of a COIP-K based protocol needs to be familiar with a few new types of data structures. One such structure is the protocol-specific structure which is used in setting performance requirements (e.g. peak bandwidth, average bandwidth, etc.) for a connection. This structure is considered protocol specific, so it is not

discussed here. The other new structures common to all COIP-K based protocols are the structures used to build a list of addresses and port numbers for a host.

All addressing information an application uses is stored in a `sockaddr_cin` structure. A `sockaddr_cin` is defined to have an address family and a list of `cinmad` structures. A `cinmad` structure contains an IP address, a port number, and an offset field (which is used internally). Figure 3.1 shows the formats of both these structures, and how a socket address can be initialized for a point-to-point connection. Note that COIP-K assumes that IP addressing will be used (thus the `in_addr` structure in the `cinmad` structure). This allows COIP-K to ignore issues such as address resolution (ARP) by allowing the normal IP code to handle it. Removing COIP-K's IP address assumption would require non-trivial changes to COIP-K's core code.

```
struct cinmad {
struct in_addr mad;
u_short cin_port;
u_short coff;
};

struct sockaddr_cin {
u_short sa_family;   /* AF_COIP */
struct cinmad cin_addr; /* PTP */
};

struct sockaddr_cin c;

c.sa_family = AF_COIP;
c.cin_addr.mad.s_addr = remote_ip_address;
c.cin_addr.cin_port = remote_port;
c.coff = 0; /* internal use */
```

Figure 3.1: COIP-K socket address structure

For a multipoint connection, the setup of the `sockaddr_cin` is more complex since the size of the socket address depends on the number of hosts in the multipoint connection. The format of the `sockaddr_cin` structure does not change, but there can be

a number of `cinmad` structures appended to it. Given a variable number of hosts in a multipoint connection, it is best to dynamically allocate space for the addresses using `malloc()`. Figure 3.2 shows an example of this. Note that the multiple `cinmad` structures are treated as an array. Treating the multiple structures as an array simplifies the programming involved in setting up multipoint connections.

```
struct cinmad *cmd;
struct sockaddr_cin *c;

c = (struct sockaddr_cin *)
malloc(sizeof(*c) + ((n - 1) * sizeof(*cmd)));

cmd = &c->cin_addr;

c->sa_family = AF_COIP;

cmd[0].mad.s_addr = remoteIP_0;
cmd[0].cin_port = remoteport_0;
cmd[0].coff = 0;
cmd[1].mad.s_addr = remoteIP_1;

/* etc. to cmd[n-1] */
```

Figure 3.2: COIP-K multipoint socket address structure

### 3.2.2. CLIENT SETUP

The COIP-K client starts by creating a COIP-K socket as shown in Figure 3.3. This creates a COIP-K protocol socket. Before this socket can be connected to a remote host, the performance requirements for the connection must be specified. Since there is no standard socket system call to do this, it is done with a `setsockopt()` system call (`setsockopt` is sort of the "catch all" socket system call). Specification of performance requirements is considered to be a protocol-specific issue, and each COIP-K based protocol is expected to define its own structure to specify such requirements. Once the application has set up this structure it can call `setsockopt()`.

After the performance requirements have been set, the application needs to set up a `struct sockaddr_cin` with the address (or addresses) of the remote host(s), as

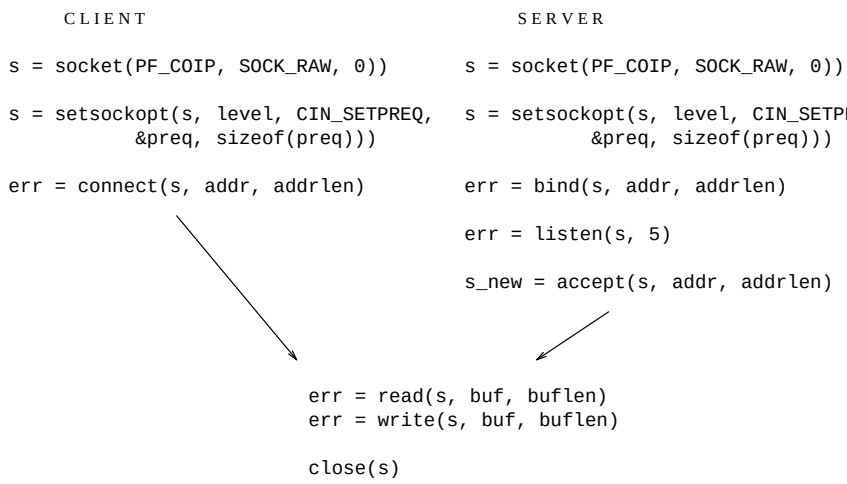



Figure 3.3: Sample COIP-K point-to-point client and server

described in the previous subsection. This is followed by a call to the `connect()` function. If the `connect()` is successful, the application is free to start I/O on the COIP-K socket using the standard read and write system calls. When the client is done with the socket, it can use `close()` to terminate the connection. If the `connect()` fails, it returns $-1$ and the socket can then be closed.

Some protocols provide mechanisms for performing control operations on a connection that has already been established. Examples include addition and deletion of a host to an already established multipoint connection. COIP-K currently does not provide support for this kind of control operation, although it can be added to COIP-K with the specification and implementation of appropriate `setsockopt()` and `getsockopt()` system calls and the addition of protocol-specific modules that go under them.

### 3.2.3. SERVER SETUP

A COIP-K server creates a socket in the same way as a client does (with the `socket()` system call). Like a client, the server also uses the `setsockopt()` call to set performance requirements. However, while the client sets its performance requirements for the connection with the `setsockopt()` call, the server uses the call to set the maximum

performance it is willing to deliver to any client. If a client requests a performance level higher than what the server is willing to provide, then the COIP-K rejects the client's connection request without any action by the sever.

The server has an option to use the `bind()` system call to bind itself to a local port number and become a well-known service. The `bind()` call takes a simple `sockaddr_cin` and looks at the `cin_port` field.

The server then calls the `listen()` function, just as in the case of TCP/IP. It then calls the `accept()` system call to accept connection requests. This call can return some information in the `addr` field to help the server determine the source of the connection, although the exact format of this information has not been specified.

### 3.2.4. DATA TRANSFER AND CONNECTION TERMINATION

Once a connection is established an application can perform `read` and `write` operations on the socket file descriptor in the standard way. When all activity on a socket is finished, the application can call `close` on the socket file descriptor to terminate the connection. Since, by default, COIP-K uses a simple closing scheme (when one side closes, the connection is terminated), higher layer protocols may have to build a more reliable closing scheme on top of COIP-K (or it could be built into COIP-K modules).

As COIP-K does no fragmentation, an application must send data in units that are less than the network's maximum transfer unit, unless there is network-level segmentation and reassembly. Also, an application or a higher layer protocol must do its own error control.

## 3.3. COIP-K IN THE PROTOCOL LAYER

COIP-K has been designed to work within the BSD Unix networking model. COIP-K lives in the protocol layer of the SUNOS/BSD kernel and has its own communications domain, as shown in Figure 3.4. COIP-K has its domain because it defines its own family of protocols that do no fall under any of the other domains. In the protocol layer, COIP-K was designed to support multiple COIP protocols concurrently, to efficiently handle per-packet processing, and to support multipoint connections. Figure 3.4 shows that each COIP-K based protocol has its own `protosw` structure. This allows an applications programmer to interface directly to COIP-K protocols in the same way as other available protocols are interfaced. COIP-K can be run without making changes to the system call interface of the socket layer. The only socket layer changes are the addition of the definitions of a few well-known constants (to identify COIP-K) in a system header file and an additional header file to define the COIP-K addressing data structures. The network interface layer must also be changed to understand the COIP-K ethernet type.

The COIP-K system can be divided into two main parts as shown in Figure 1.1. The first part is the core COIP-K code which is common to all COIP-K protocols. The second part is the group of protocol-specific modules which are plugged in on top of the core code to form an implementation of a COIP protocol. A COIP protocol built with COIP-K should support connection-oriented[1] communications with resource allocation, packet forwarding/gatewaying, and multipoint connections. The main assumption that the COIP-K code makes is that IP addresses will be used. Also, by default, COIP-K uses IP routing. This can be overridden if the need arises.

[1]Note that the "connection" is not a reliable connection. Instead, it is a combination of a reliable connection and a datagram, thus it is sometimes called a "congram."

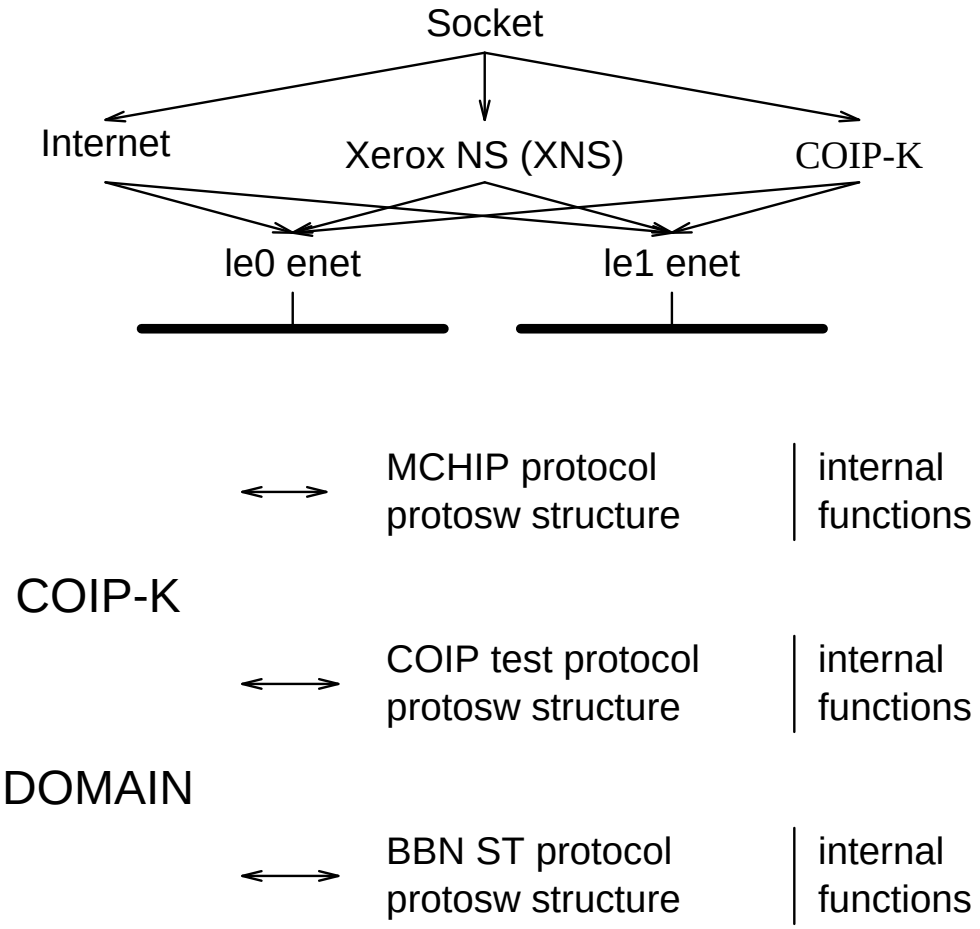



Figure 3.4: The BSD Unix networking model and COIP-K

### 3.3.1. DATA STRUCTURES - THE COIP-K PCB

The most important data structure of COIP-K is the COIP-K protocol control block (PCB) which is shown in Figure 3.5.

The COIP-K PCBs are stored in a circular linked list which can be traversed by starting with the address of a dummy PCB `cin_q` and following the `p_next` pointer. The PCB structure is set up so that the standard routines `insque()` and `remque()` can be used to link and unlink PCBs from the active list. Note that each end-point of a connection has its own PCB structure associated with it.

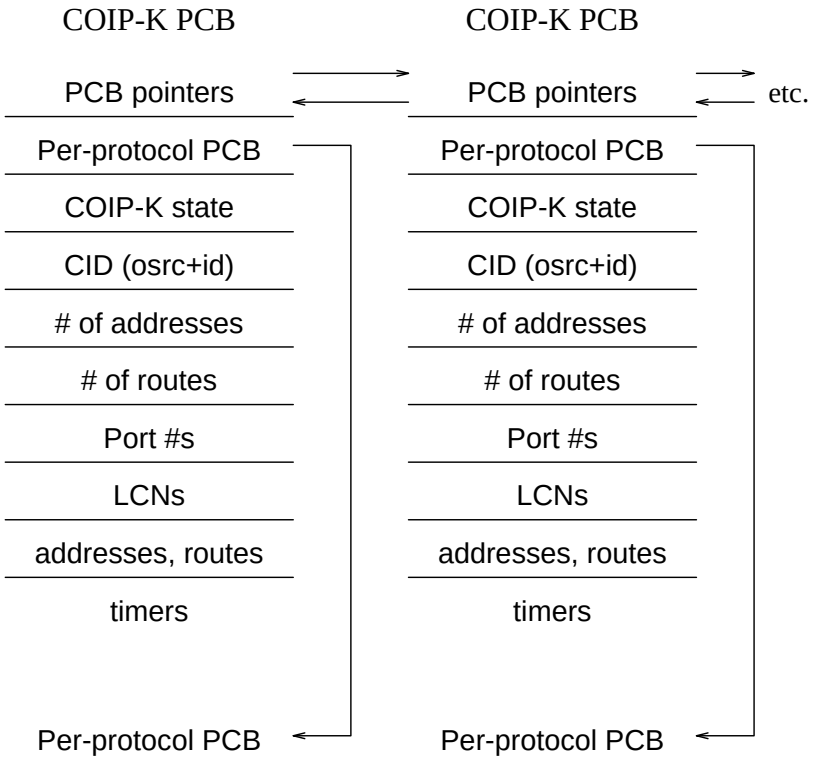



Figure 3.5: COIP-K PCB Structure

In COIP-K, a per-protocol control block is an mbuf which is used to store protocol-specific state information. Because the information in this mbuf is protocol specific, its structure is not defined by COIP-K. The pointer to the per-protocol control block resides in the COIP-K PCB. The per-protocol control block must be allocated and released at the same time as the main PCB. Thus, the user request function will call a protocol-specific module every time it makes or destroys a COIP-K PCB.

The COIP-K state variable indicates to the COIP-K the state of the corresponding connection (e.g. CLOSED, OPEN, OPENING, etc.). Protocols which require additional state information can use the per-protocol control block to store that information. The ID numbers in the COIP-K PCB are the connection identifiers (CIDs) and the logical channel numbers (LCNs). The CID consists of a unique eight byte number which distinguishes the connection from all other connections on the network. The first four bytes are the

IP address of the host which originated the connection. This information is called the `osrc` (originating source). The second four bytes are a unique ID number created by the originator. The CID applies to every host and gateway in the connection. The LCNs on the other hand are strictly hop-to-hop ID numbers. An LCN is two bytes in length and indicates a data flow in one direction. Thus, a full duplex connection requires two LCNs, one for inbound data and one for outbound data.

**MULTIPOINT ADDRESSING AND ROUTING.** The format of addressing and routing information stored in the COIP-K PCB depends on whether the connection is a point-to-point or multipoint connection. For point-to-point connections, all addressing and routing information is stored in the main COIP-K PCB. For a multipoint connection, addressing and routing information is stored in separate mbufs as shown in Figure 3.6. This set up was chosen because it fits in easiest with the limitations of the mbuf system. Currently, the addressing and routing mbufs limit the size of the data to 1024 bytes each, but this restriction can be removed if larger data sizes are needed in the future. The structure of these two mbufs is described in the following paragraphs.

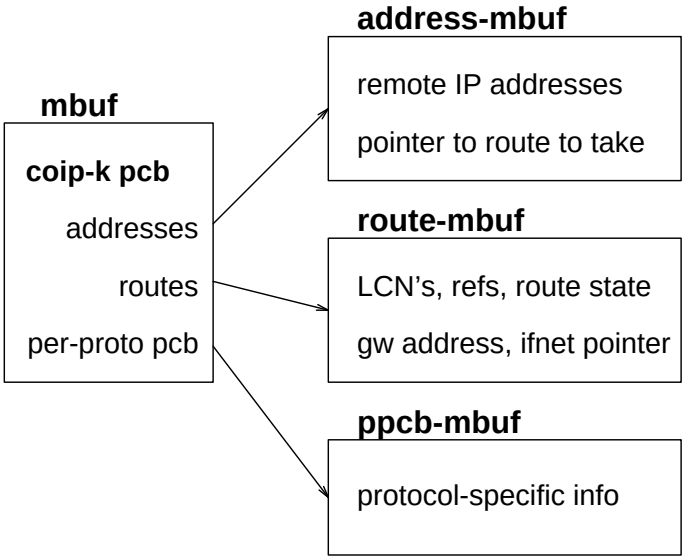



Figure 3.6: Multipoint PCB structure

Addresses of a multipoint connection are stored in the mbuf **cinmad-mbuf** which consists of a number of `cinmad` structures. Each structure contains an IP address of one of the destinations and a pointer to a route for reaching the remote host. The routing information pointed to by the `cinmad` structure is stored in the routing mbuf and consists of `cinmdst` structures. It is possible for several different addresses to point to the same `cinmdst` route. A `cinmdst` structure consists of:

- the input and output LCNs for this route
- the number of addresses (references) which use this route
- the COIP-K state of this route
- the remote address (needed for ethernet destination if this route is not through a gateway)
- a `rtentry` structure that is used to access the network interface which is associated with this route.

In testing COIP-K, the `rtentry` structure was obtained from the IP routing system, however, COIP-K will not force protocols to use standard IP routing. For example, a COIP-K protocol could have its own separate and private routing table that maps IP addresses to network interfaces. This table, which is separate from IP, would be accessed via COIP-K modules.

### 3.3.2. MAJOR FUNCTIONS

Before considering the details of connection establishment, data transfer, and connection termination, we shall first present an overview of the main functions involved. Figure 3.7 shows how a COIP-K protocol fits in with the other layers. Note that the actual

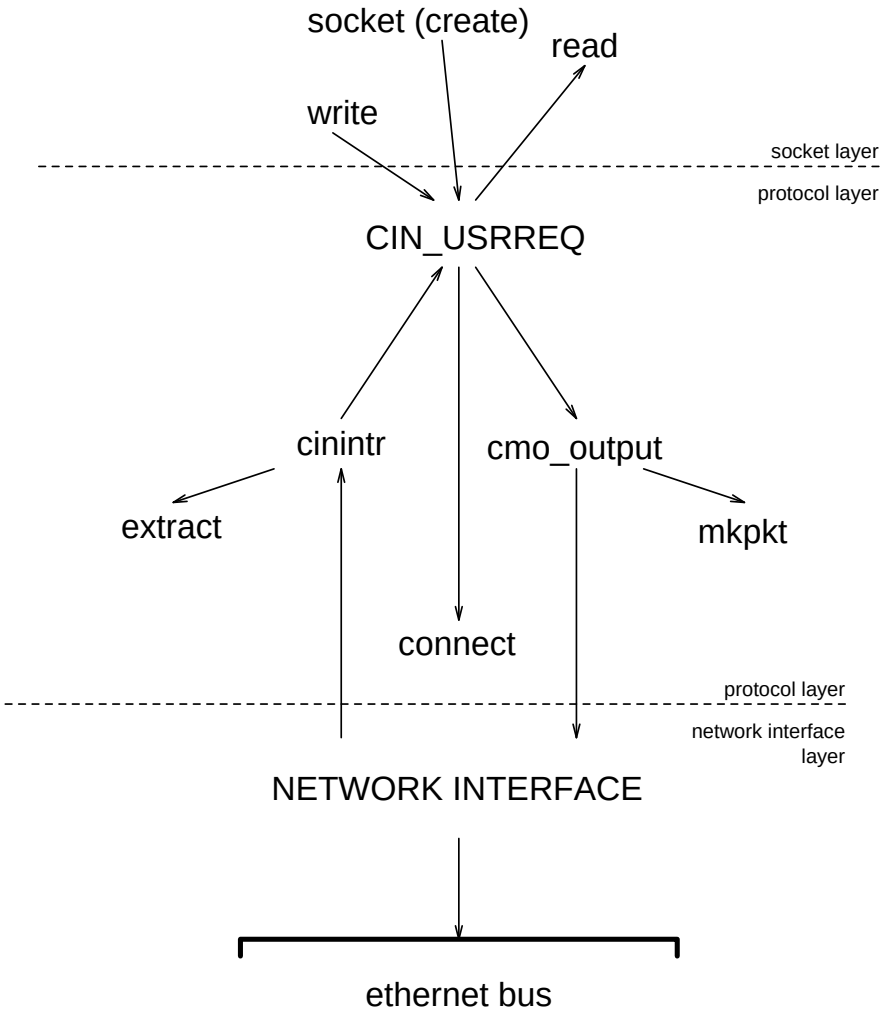



Figure 3.7: COIP-K protocol switch

COIP-K implementation consists of a large number of functions which (for the sake of clarity) are not shown in Figure 3.7. The functions labeled "extract" and "mkpkt" are actually required protocol-specific COIP-K modules and are described in Section 3.4.1. Note the similarity between the COIP-K protocol switch shown in Figure 3.7 and the TCP/IP protocol switch shown in Figure 2.10.

**USER REQUEST FUNCTION.** The user request function `cin_usrreq` is the socket layer's interface to COIP-K. This function does many different tasks such as

socket/PCB creation, the processing of socket options, reads and writes, connection establishment for the client side, local-address binding, and setting up a PCB to accept inbound connections.

The user request function is called from the socket layer. Among its arguments are type of request being made and the socket associated with the request. The user request function first looks up the PCB of this socket. If the socket has just been created, then it will have no PCB and the user request function will create a new PCB for it. The user request function then switches on the request argument and processes it. Finally, it returns control to the socket layer.

A general outline (in pseudo-code) of the user request function is as follows:

```
cin_usrreq(socket, request, mbuf, nam, rights) {
  struct cinpcb *pcb; /* a COIP-K PCB */
  /* the request parameter is set to the request type */
  pcb = pcb pointer from ‘‘socket’’; /* get pcb, if there is one */
  if (request == PRU_ATTACH && pcb == NULL)  /* no pcb? */
    create new pcb;
  else
    return an error code;
  switch (request) {    /* switch on request */
    case PRU_SEND: /* send data */
      do send stuff
    break;
    case PRU_CONNECT: /* connect */
      do connect stuff
```

```
    break;
    etc...
    default:
      error condition
  }
  return control to socket layer;
}
```

**INTERRUPT FUNCTION.** The `cinintr` function, the network interface layer's interface to COIP-K, is called by the network interface layer when a COIP-K packet is received. This function performs tasks such as packet forwarding, data input (from the network), and the server side of connection establishment.

The interrupt function is scheduled to be called by the network interface layer. It is basically a loop in which a packet is removed from the input queue and processed until the input queue is empty. A packet is processed by first determining its COIP-K protocol and packet type. Then, the LCN is extracted, and the PCB is looked up. Finally, the packet is processed as either a data, open, resource, or control packet. The pseudo-code for the `cinintr()` function is:

```
cinintr() {
  struct mbuf *packet;  /* a packet */
  struct coip_proto *cmo; /* coip-k module set pointer */
  struct cinpcb *pcb;
  top:
    mbuf = the first packet on the COIP-K queue;
    if (mbuf == NULL) return;   /* finished cinintr */
```

```
cmo == NULL;
for each COIP-K module set
  if (packet type function claims packet) {
    cmo = current module set;
    break;
  }
if (cmo == NULL) {
  free packet buffer;    /* drop packet */
  goto top;
}
extract LCN from packet;
pcb = result of PCB lookup module;
if (packet is data packet) {
  process packet, forward packet;
  goto top;
}
if (packet is open packet) {
  process packet with connection establishment code;
  goto top;
}
if (packet is control packet) {
  pass packet to control input module;
} else if (packet is resource packet) {
  pass packet to resource allocation module;
}
forward packet if needed;
```

```
    goto top;
}
```

**OUTPUT FUNCTION.** The packet output function, `cmo_output()`, takes four arguments. The first argument is the PCB pointer, which is used to get routing information and data from the PCB. The second is the packet type, which determines what type of packet `cmo_output()` asks the COIP-K modules to make. The last two arguments are an indication of what interface the packet came in on and what the LCN was. If both are 0, then the packet originated from the host, otherwise it was forwarded. The `cmo_output` function handles forwarding by sending data to every point on the connection except the point the data came in on.

**CONNECT FUNCTION.** The connect function is illustrated in Figure 3.8. The function takes a list of addresses and a PCB, and using the protocol-specific routing module and the LCN mapping module it produces a PCB with all the routing and addressing information set up.

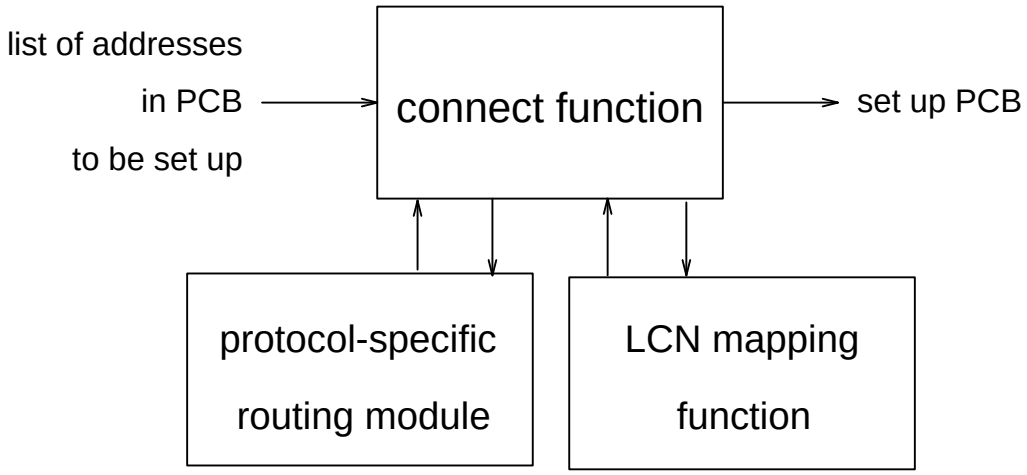



Figure 3.8: Connect function

**CONNECTION MANAGEMENT.** COIP-K, by default, provides only a very basic connection management scheme. It assumes that all endpoints of a connection are known at connect time, and that they can not be added or deleted after a connection is established. Also, `coip-k`'s default concept of connection establishment is not absolutely reliable or efficient. It depends on a simple two-way handshake and timers to detect errors. Another limitation of default COIP-K behavior is that COIP-K uses a simple linear search to find a PCB given a packet. In order to support a large number of connections a more sophisticated hashing search would have to be used. To provide more elaborate connection management, more complex protocol specific functional modules must be provided.

## 3.4. COIP-K MODULES

There are two types of COIP-K modules: required and optional. Required modules are ones that are totally protocol specific and must be provided by the protocol implementer. Optional modules are modules that may or may not be provided by the protocol implementer. If they are not provided, COIP-K provides a reasonable default module from its module toolbox. This set up is shown in Figure 3.9. By providing default toolbox modules, COIP-K provides an incremental level of support for protocol programmers. Novice kernel programmers can use mostly toolbox modules and get something running quickly, and as they become more advanced they can swap out toolbox modules for more advanced modules of their own. The next two subsections list the COIP-K modules.

### 3.4.1. REQUIRED MODULES

**Extract module:** COIP-K does not know a packet's format because it is a protocol-specific detail. Therefore, when COIP-K must remove vital bits of information from

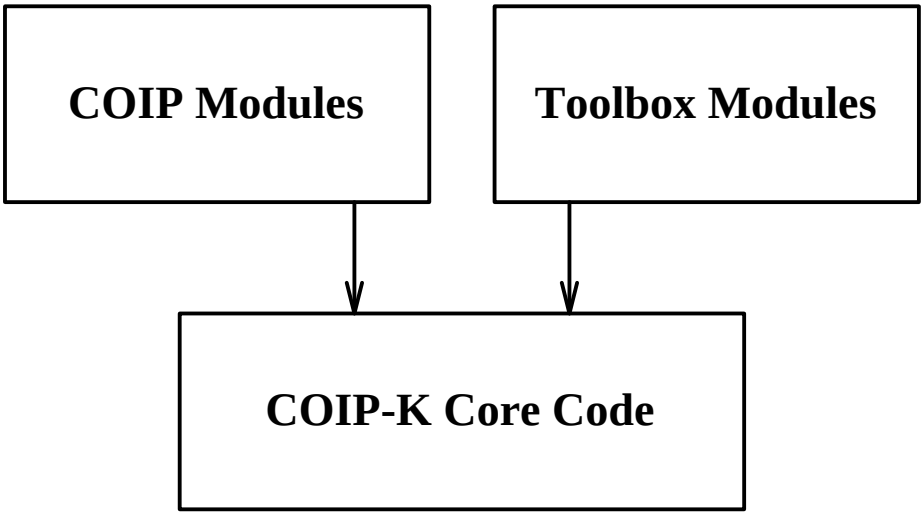



Figure 3.9: COIP-K module plug in

a packet, it uses the extract module. This interface is similar to the information-hiding techniques used in object oriented programming.

**Make packet module:** As with the extract module, COIP-K does not know a packet's format. Thus, the make packet module is required in order to form a packet from data.

**Packet type module:** The packet type module determines if a packet is associated with a module set. If the packet is recognized by a module set, then the packet type module will return the packet's type. If the packet is not recognized, then the packet type module returns an error code. This module is first called in the COIP interrupt function to determine which module set to use when deciding a packet's fate.

**Pcb lookup module:** When a packet is received, the COIP-K protocol is determined (by using the packet type module). Then the PCB lookup module is used to determine which PCB the packet is associated with.

### 3.4.2. OPTIONAL MODULES (TOOLBOX MODULES)

**Attach module:** If this module exists, it is called at PCB creation time to initialize any protocol-specific data structures such as the per-protocol-PCB.

**Connect module:** An application can request that a socket be connected to a list of hosts with the `connect()` system call. After receiving a `sockaddr_cin` COIP-K calls the connect module with the `sockaddr_cin` structure. If present, the connect module is expected to setup routes to all the addresses in the structure. If no connect module is present, COIP-K will use its own internal connect function with standard IP routing.

**Control input module:** When packets arrive on a COIP-K connection they are either data, resource, or control packets. The control input module, called from the `cinintr()` function, takes a control packet and processes it. The default control input module from the

sc coip module toolbox understands acknowledgments to open and close packets. For more complex protocols, the default module can easily be replaced.

**Data input module:** When a data packet is received by COIP-K in `cinintr()` it can do one of two things. It can either pass the data directly to the socket layer, or it can pass it to the data input module for further processing (e.g. error detection or a higher level protocol). The default is to not provide a data input module so that the data is passed directly to the socket.

**Detach module:** If this module exists, it is called right before COIP-K frees a COIP-K PCB so that any protocol-specific resources allocated by a protocol (typically in the attach module) can be freed first.

**Disconnect module:** When a COIP-K connection terminates it must free up any system resources that it has allocated (e.g. routes, buffers). The disconnect module handles this task.

**Fast timer module:** This function is called every 200 ms if present. It provides a way for a COIP-K protocol to do fast timeouts.

**Init module:** Many protocols need to set up data structures and timers when the system is first booted. The init module provides a way for the protocols to do this. Unlike the timer modules, the init function is located in the standard `protosw` structure.

**Localized module:** The localized module determines if the address of the current machine is in a list of addresses. It is useful for determining if a packet is intended for the local machine or not.

**Output module:** The output module is responsible for outputting data to the network interface layer. The standard output module was described previously in Section 3.3.2.

**Pcb setup module:** This function is related to the connect function, except it is called when an OPEN packet is received instead of when a `connect()` system call is used. If it is not present, the COIP-K default function (with IP routing) is called.

**Performance packet input module:** This function is called by the interrupt routine to handle performance packets from a resource server or remote host. It is similar to the control input function, except it expects packets relating to resource allocation. If the function is not present, resource packets are dropped. (It should be noted that resource packets are really a type of control packet, but COIP-K

separates the two packets into different classes to make the resource allocation function more modular.)

**Reject module:** The reject module is used to send an error message to a peer COIP-K system. Currently error conditions are not well defined, so this function is not used much.

**Set performance requirement module:** This function is called from `setsockopt()` to set the performance requirements of a connection. If it is not present then performance requirements for the protocol are turned off.

**Slow timer module:** This function is called every 500 ms. This provides a way for COIP-K protocols to do slow timeouts. Slow timeouts must be used to timeout the `connect()` system call. If a slow timer module is not provided, the default one will perform this task.

# 4. COIP-K FEASIBILITY AND VIABILITY

## 4.1. INTRODUCTION

This chapter presents the feasibility and viability of COIP-K, which has been designed to meet four main objectives:

- COIP-K should facilitate the implementation of different COIPs.
- COIP-K should allow COIPs to be easily constructed by interchanging modules. The original model specified that a set of modules could be compiled into the kernel with COIP-K to form a single COIP protocol. As COIP-K was developed, the scope of this model was revised and enhanced, as will be seen in this chapter.
- COIP-K should provide support for multipoint communication.
- COIP-K should provide efficient per-packet processing (either as a gateway or as an endpoint).

The purpose of this chapter is to demonstrate how these objectives have been successfully achieved, and in some cases surpassed. The outline of the chapter follows.

Section 4.2 presents a specification of a simple COIP protocol, the COIP Test Protocol (CTP), and shows how its implementation was realized using COIP-K. This exercise has served two purposes. First it has helped demonstrate COIP-K's usefulness in creating implementations of a COIP protocol. Second, it has helped us thoroughly debug and test COIP-K. It is important to note that this simple COIP protocol represents a subset of Washington University's MCHIP protocol.

Section 4.3 presents important aspects of the COIP-K organization which facilitate module interchange and incremental support. Incremental support allows novice COIP-K programmers to start off relying on COIP-K to do most of the hard work. Then, as novice COIP-K programmers become more advanced, they can use incremental support to rely less on COIP-K code and more on their own code. Easy and efficient module interchange is critical in realizing COIPs using COIP-K and in comparing alternate solutions of COIP modules. We show by example that the COIP-K organization does indeed make the module interchange easy.

Section 4.4 presents performance results of COIP-K obtained through a series of experiments. The purpose of these experiments is to characterize the per-packet processing effectiveness of COIP-K and to also quantify the cost of the COIP-K concept. The cost of COIP-K is defined as the additional overhead of packet processing using COIP-K as compared to direct protocol implementation. This is quantified in terms of the number of additional function calls and protocol-independent tasks needed to do per-packet processing.

Section 4.5 presents a number of demonstration applications created on COIP-K. These demonstrations serve three objectives. First, they test and verify several capabilities (point-to-point, multipoint, gatewaying, etc.) of COIP-K. Second, they show that applications using the standard socket interface can be ported to work on COIP-K with minimal effort. Finally, these applications show that COIP-K can be used to create useful applications.

## 4.2. COIP-K TEST PROTOCOL

The COIP-K Test Protocol (CTP), a COIP protocol implemented using COIP-K, has been used to test COIP-K. The CTP protocol has been intentionally kept simple because our

emphasis is on COIP-K. We expect CTP to serve as a template for implementation of other COIP protocols using COIP-K.

One main difference between CTP and MCHIP is that MCHIP allows resource reservations to provide performance guarantees. Although COIP-K has been designed to support resource allocation and enforcement modules, we did not implement this in CTP. We believe resource allocation and enforcement are protocol specific, and thus should be part of COIP modules and not part of COIP-K itself.

### 4.2.1. CTP PACKET FORMATS

This section presents the CTP packet formats. All CTP packets start with a standard CTP header. This header consists of a one byte version number, one byte of padding, and a two byte packet type.

**CTP OPEN PACKET FORMAT.** The "open" packet format is shown in Figure 4.1. The open packet starts with the standard four-byte CTP header, with the packet type set to `OPEN`. The next field in the packet is the IP address and COIP port number of the source host[1]. (Port numbers are needed to distinguish between COIP-K connections between the same machines.) After the COIP source port number there are two bytes of padding. The packet then has two fields which together produce a unique eight-byte identification number for the connection. The exact structure of this field is not of major importance as long as the eight bytes are unique across the network. In CTP, we decided to have the first four bytes contain a portion of the IP address of the host which originated the connection (called the originator or OSRC), and the second four bytes contain a unique number generated by that host (called the connection ID or CID). The next field is the LCN, or logical channel number. The LCN is a two-byte

[1]Note that a host can act as a gateway and/or an endpoint in a COIP-K connection.

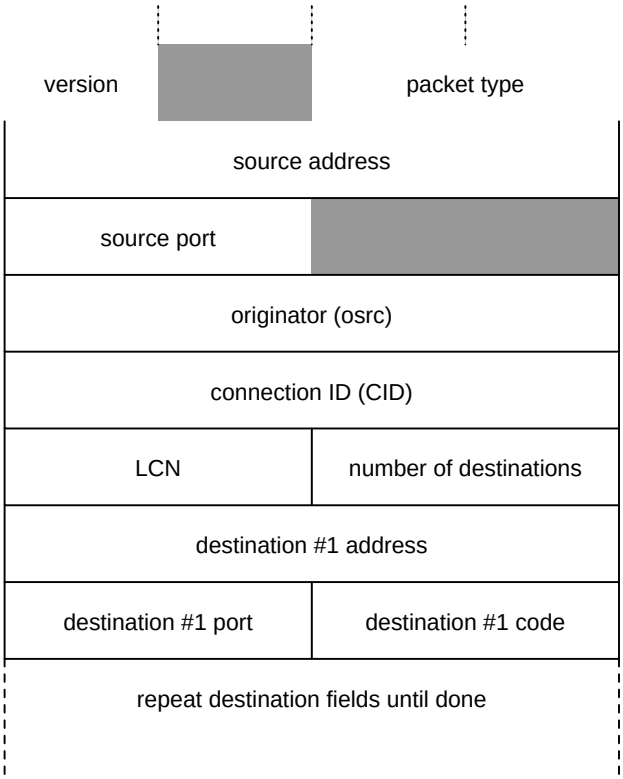



Figure 4.1: Open packet

number which is used between two adjacent hosts on a COIP connection to indicate which session the packet belongs to. After the connection is established, only the LCN is used to identify the connection. The next field indicates the number of hosts associated with this connection. The rest of the open packet consists of information on each of the hosts in the connection as shown in Figure 4.1. By including this information in the open packet, each endpoint in the connection can get a list of all the other endpoints on the connection.

For each host in the connection, a three-field structure is appended to the open packet. The first field is the IP address of the destination host. The second field is the COIP port number to connect to on that host. The third field is a "code" which indicates the disposition of that destination on the connection. There are three possible values for the code and they are:

- `COF_IGNORE` (0xffff) — ignore address
- `COF_PARENT` (0xfffe) — address is parent in connection tree
- 0 (zero) — normal address

The host or gateway that receives the open packet uses the code to determine if it needs to connect to a destination or if that destination is already connected at some other end of the connection. If the value of a destination's code is `COF_PARENT`, the destination is the parent of the current host in the connection tree. If the value of a destination's code is `COF_IGNORE`, the destination is already on the connection (from some other end) and the host which has received the open packet does not need to take any action to connect to that destination host. Finally, if the value of a destination's code is 0, either the destination is on the host which has received the open packet, or the host which received the open packet is to act as a gateway to that destination. Figure 4.2 shows an example of values of the code for a multipoint connection between hosts A, B, C, D and O.

Note that the three-field destination structure mirrors the COIP-K `cinmad` structure.

**CTP ACK OPEN PACKET.** Figure 4.3 shows the format of the "ack open" packet. This packet has the usual four-byte header, with the packet type set to "ack open." This is followed by the LCN which will be used by the remote host when sending data packets. It then has both the LCN that the host who sent the open packet used, and the CID. These allow the host to double check the references before the connection is marked as open.

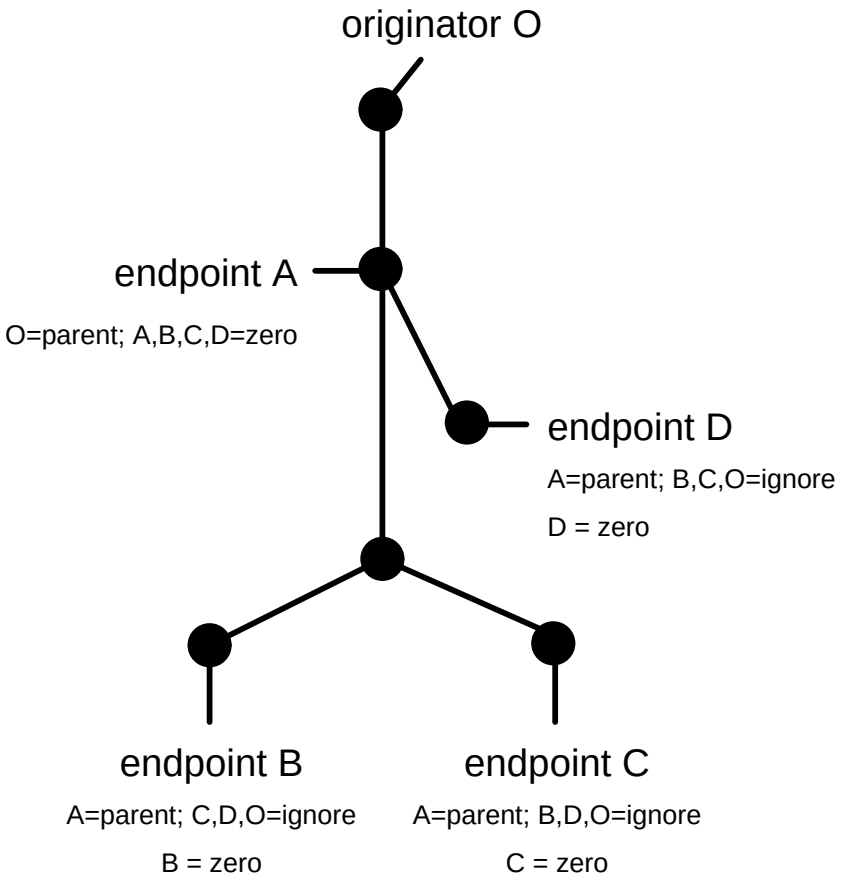



Figure 4.2: Example of the use of the code field in an open packet

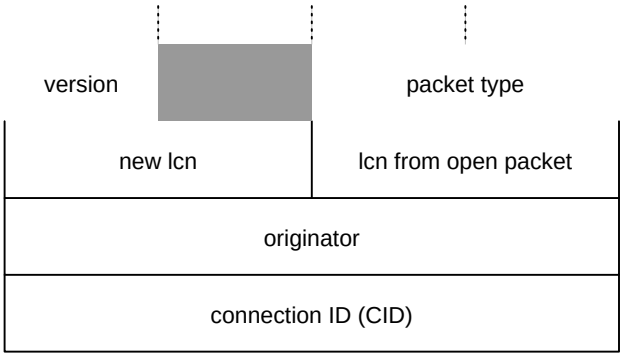



Figure 4.3: ACK open packet

**CTP DATA PACKET.** The format for a data packet is rather simple and is shown in Figure 4.4. In this example the packet type is set to "data." The only other infor-

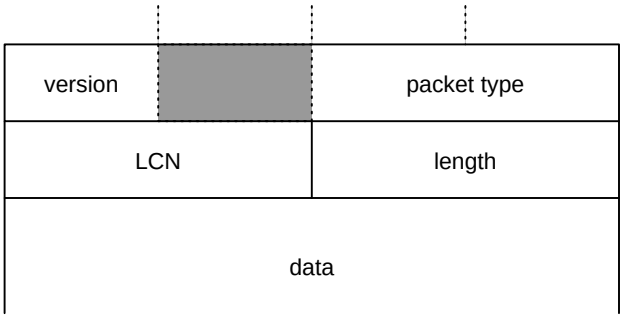



Figure 4.4: Data packet

mation in the header are the LCN and data length fields.

**CTP CLOSE AND ERROR PACKETS.** Finally, the last CTP packet format to be presented is the close and reject format. The close packet is actually a special case of the reject packet (i.e. they both have the same format). The reject packet exists so that an error message facility similar to the Internet Control Message Protocol [4] (ICMP) can be built in CTP. This facility has not been developed yet, and thus is not described here. The close packet is pictured in Figure 4.5. In addition to the standard

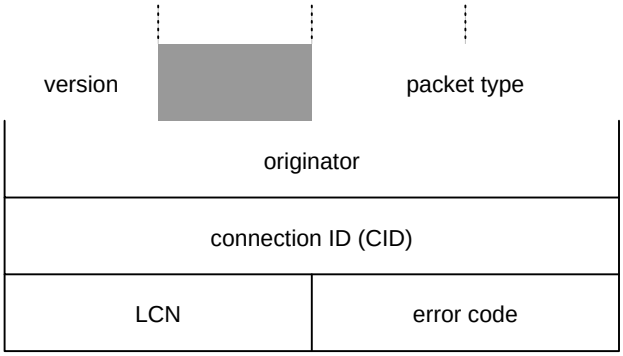



Figure 4.5: Close packet

CTP header, the close packet contains the OSRC, CID, and LCN of the connection. The

format also includes space for an error code, however CTP does not make use of this space at this time.

### 4.2.2. CTP CONNECTION LIFELINE

This section gives a general overview of the life of a CTP/COIP-K connection.

**CTP CONNECTION ESTABLISHMENT.** Connection establishment in CTP is shown in Figure 4.6. The COIP-K states are shown on the sides of the vertical time

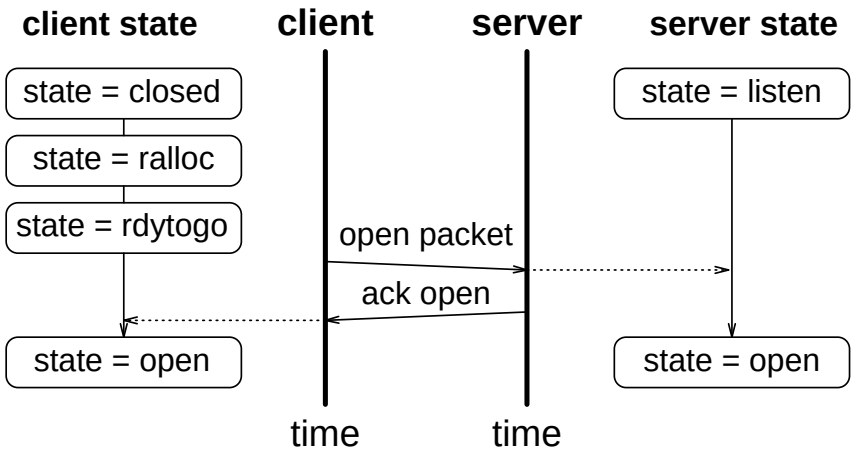



Figure 4.6: Connection establishment

lines, and the packets are shown in between the lines. This connection could be a point-to-point connection, or it could be one branch of a multipoint connection.

When the client process sends the server an open packet, the server replies with an ack open packet opening the connection.

**CTP DATA TRANSFER.** Once the connection is established, data can be transferred until a close packet is sent. This is shown in Figure 4.7. The packets are all in the CTP data format.

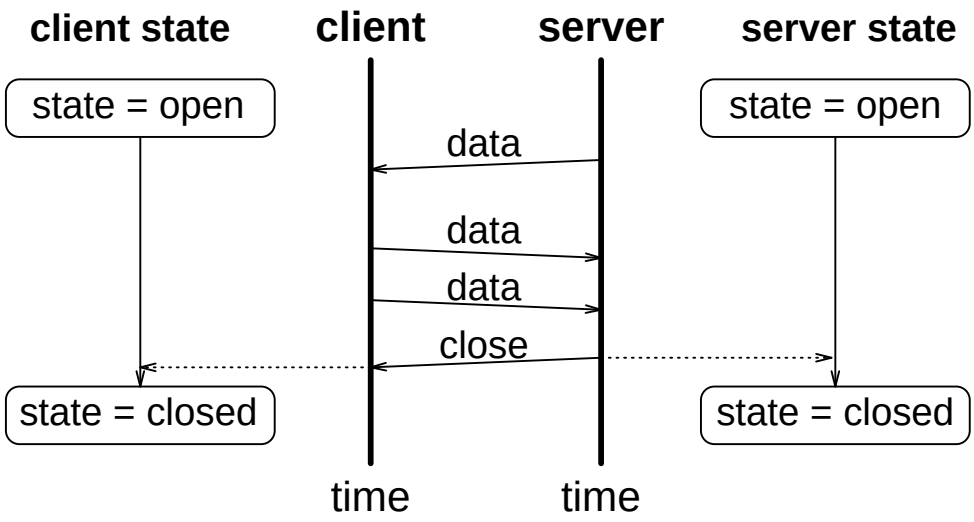



Figure 4.7: Data transfer and connection closing

**CTP SHUTDOWN.** Shutdown is also shown in Figure 4.7. When one side sends a close packet, the connection is considered closed. Since this type of connection closing is not reliable, a higher level protocol should be used to make sure that all end points in the connection are ready to close before closing the COIP-K connection.

### 4.2.3. COIP-K MODULES REQUIRED FOR CTP

Because CTP is a simple COIP protocol, it required only four modules to implement. Only required modules are provided with CTP. For optional modules, CTP uses the default modules from the COIP-K module toolbox.

**CTP EXTRACT AND MAKE PACKET MODULES.** The extract and make packet functions provide COIP-K with access to the CTP packet format. The extract function takes a CTP packet and extracts various data fields from it. The make packet function takes data and creates a CTP packet from it. The exact CTP packet format is described in Section 4.2.1.

**PCB LOOKUP MODULE.** This CTP function takes any packet and looks for a PCB associated with it. In CTP this means doing extracts based on the packet type and then either looking through the COIP-K PCBs or doing an LCN lookup.

**PACKET TYPE MODULE.** This module takes as input a COIP-K packet and determines if the packet is a CTP packet, and if so, what type of packet it is. If the packet is not a CTP packet, the packet type module returns 0 which indicates the packet belongs to some other COIP protocol.

### 4.2.4. CTP AND COIP-K TRACES

Though the external interface to CTP looks simple, its implementation with COIP-K is reasonably complex. This subsection presents connection traces for connection establishment, data transfers, and connection termination (in that order). A connection trace of an action is a description of the functions called when that action takes place. Connection traces can help a programmer understand the complexity and details of the code. The figures in this section denote two types of modules. Functions which start with "`cmo_`" are part of the COIP-K toolbox. Functions which start with "`ctp_`" are CTP-protocol-specific modules.

**CONNECTION ESTABLISHMENT.** The connection establishment phase of a connection is the most complex aspect of COIP-K. A basic outline for the client side of the connection establishment is as follows:

1. user request function gets list of addresses to connect to
2. local port number is bound (if needed)
3. connection function is called to set up a PCB

4. client sends OPEN packet

5. process sleeps waiting for ACK(s) in response to the OPEN packet(s)

From the application programmer aspect, this process is shown in Figure 3.3.

The `cinintr()` routine receives OPEN packets and sends a response. It also receives the ACKs of open packets and wakes the sleeping process when the connection is opened.

We shall now examine the connection establishment of a point-to-point connection using the CTP protocol. Note that resource allocation is not done in CTP. The connection establishment trace is shown in Figure 4.8.

**SERVER START UP.** After the server is started, it calls `socket()` to create a COIP-K socket. This calls the `cin_usrreq()` function with the `PRU_ATTACH` directive which creates a PCB for the socket and attaches CTP to it. It then returns control to the program. The program subsequently initializes a `sockaddr_cin` structure and calls `bind()` to set its local address to a well-known service address. This causes `cin_usrreq()` to be called with the `PRU_BIND` directive. The user request function calls the `cin_lbind()` which checks to make sure the address requested is not in use by checking all the PCBs in the active list. If the address is free, the function sets the PCB of the socket to that address and returns control to `cin_usrreq` which returns control to the application. The server then calls the `listen()` function which causes the `cin_usrreq()` function to be called with the `PRU_LISTEN` directive. The user request function changes the COIP-K state of the socket to LISTEN and then returns control to the user. The user then calls `accept()` which causes the socket layer to put the user process to sleep until a connection request arrives.

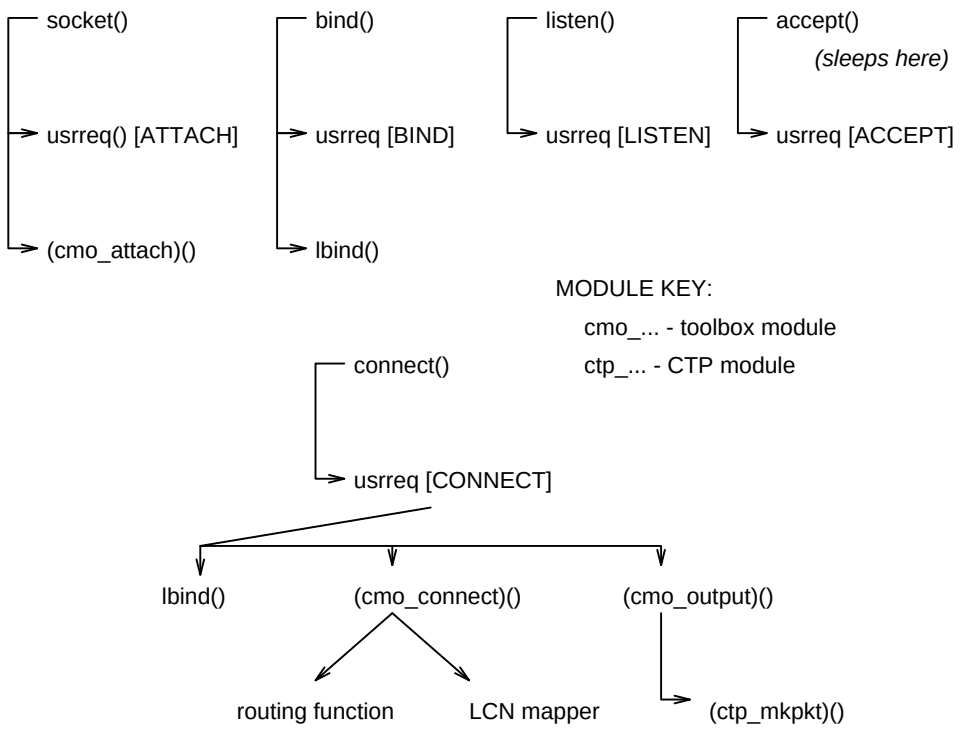



Figure 4.8: Server (top) and client (bottom) connection establishment

**CLIENT START UP.** In the mean time the client starts. It calls `socket()` in the same way as the server does. It then sets up a `sockaddr_cin` and calls `connect()` to connect to the remote host. This causes the `cin_usrreq()` function on the client to be called with the `PRU_CONNECT` directive. The user request function then calls `cin_lbind()` to assign the client a random port number, as shown in the bottom half of Figure 4.8. It then calls the `cmo_connect()` function to set up the PCB. This connect function is a part of the COIP-K toolbox which takes a list of addresses to connect to and sets up the routing and addressing information in the PCB. It does this by calling both the routing module (the IP router in the test protocol's case) and the LCN mapping module. The LCN mapping module maps LCNs (which are shared by all COIP-K protocols) to PCBs. Once a LCN has been bound to a PCB, all reference to that connection will be via the LCN. If the PCB is needed, the LCN module can do a quick table lookup based on the LCN. Once a PCB has been setup by the connect function, it is possible to use the routing

information to output packets on the connection. In the OPEN case, after calling the connection function, the `cin_usrreq()` function calls the protocol-specific packet construction module. This module is responsible for creating packets by allocating an mbuf to store the packet and then setting up fields according to the type of packet requested. Once the open packet has been created, the user request function calls the `cmo_output()` function which takes the open packet and sends it to the remote end. Then the user request function returns control to the socket layer which puts the process to sleep until either a timeout is received or the connection is established.

**RECEPTION OF OPEN PACKET BY SERVER.** The next step in the connection establishment process occurs when the server receives the OPEN packet as shown in Figure 4.9. As a result, the `cinintr()` function is called. This function first dequeues the packet from the network interface queue and then determines the type of the packet and the COIP-K protocol it belongs to. Since this is an OPEN packet, the interrupt function transfers to the OPEN case. The PCB of the listening socket (if any) is looked up. This is done with a protocol-specific module which takes a packet and looks up its PCB (if there is one). Then the `ctp_extract()` function is called to extract the addressing, CID, and LCN information from the packet. The extract function is also a protocol-specific module. Once the addressing information is extracted, the interrupt function calls the `cin_localized()` function to determine if this packet is for a process on the local host, or if it should be forwarded. In the example in Figure 4.9, the packet is a point-to-point packet for the local host, so the interrupt function calls `sonewconn()` to let the socket layer create a new socket for the connection. The `sonewconn()` function calls the user request function with the `PRU_ATTACH` directive to create and link the PCB for the socket. The socket created is set to state OPEN. Then control goes back to the interrupt function which calls the `cin_allopen()` function to look at the PCB and

determine if the connection is all open or if the OPEN packet needs to be forwarded to other hosts (e.g., in the case of a multipoint connection). If the connection is all open, as in this example, then the interrupt function calls the `ctp_mkpkt()` function to create an "ack open" packet and then calls `cmo_output()` to send it to the client.

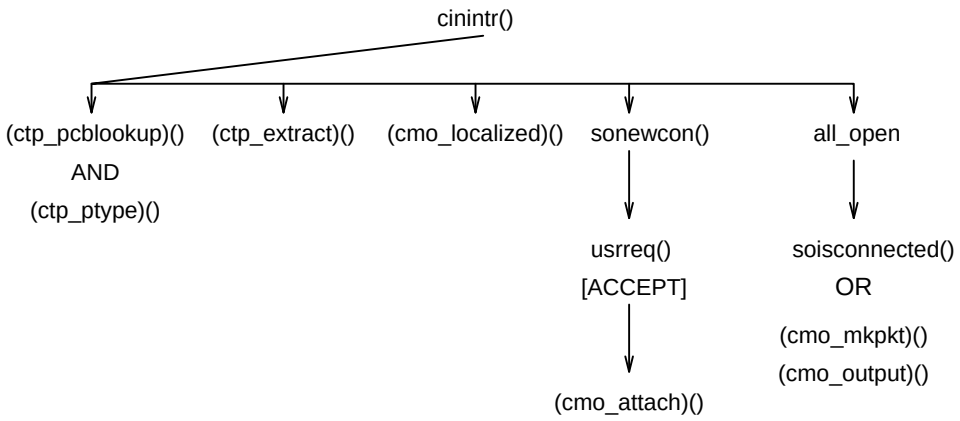



Figure 4.9: Open packet reception at the server

**RECEPTION OF AN ACK OPEN PACKET BY CLIENT.** When the "ack open" packet is received by the client's host-network interface the `cinintr()` function is called. It dequeues the packet from the interface queue and determines its protocol, packet type, and any related PCB (as above). Since the packet is an ACK to an open packet, it is passed to the control packet input module with the PCB of the client. This module reads the "ack open" packet, sets the PCB state to open, and informs the socket layer that the connection is open (with the `soisconnected()` function). This wakes up the user process from the connect system call. At this point the connection is established.

**DATA TRANSFER.** As shown by the function trace in Figure 4.10, data transfer in the COIP-K system is fairly simple. One side of the connection calls a `write()` which causes the user request function to be called with the `PRU_SEND` directive. The

user request function takes the data, calls the `ctp_mkpkt()` module, and then calls `cmo_output()` to send the data. The output function handles both output and packet forwarding. It handles the latter by taking an optional parameter which indicates which route the packet came in on. With that information it can forward the packet to all other routes with the exception of the one it came in on. Thus, multipoint connections are handled like a flooding protocol.

The `cinintr()` routine receives the data packet in the same way as the open packet is received. However with a data packet the interrupt routine simply extracts the data and either passes it to a protocol-specific data input module or directly to the socket layer if that module is not defined. This is done with the `sbappend()` and `sorwakeup()` functions. The work of the `read()` system call is mostly handled by the socket layer. This process is illustrated in Figure 4.10. Then, if the host is acting as a gateway, the data packet is forwarded to the next hop.

It is important to note that with COIP protocols, all routing and resource allocation is done at the time of connection establishment. Once the connection is set up, all that is needed from the data packets is an inbound LCN number. This allows the critical data transfer portion of the protocol to be implemented in hardware [7]. Figure 4.11 shows data transfer through a gateway and the LCN's required for data transfer.

**CONNECTION TERMINATION.** A COIP-K connection termination is started with the `close()` system call, which causes the system to call `cin_usrreq()` with the `PRU_DISCONNECT` directive. This causes COIP-K to send a CLOSE packet to the remote system. Then the `cin_usrreq()` is called again, this time with the PRU_DETACH directive. This causes COIP-K to delete the PCB of the socket and mark the file descriptor as closed. On the server side, the client gets the packet in `cinintr()` and calls the

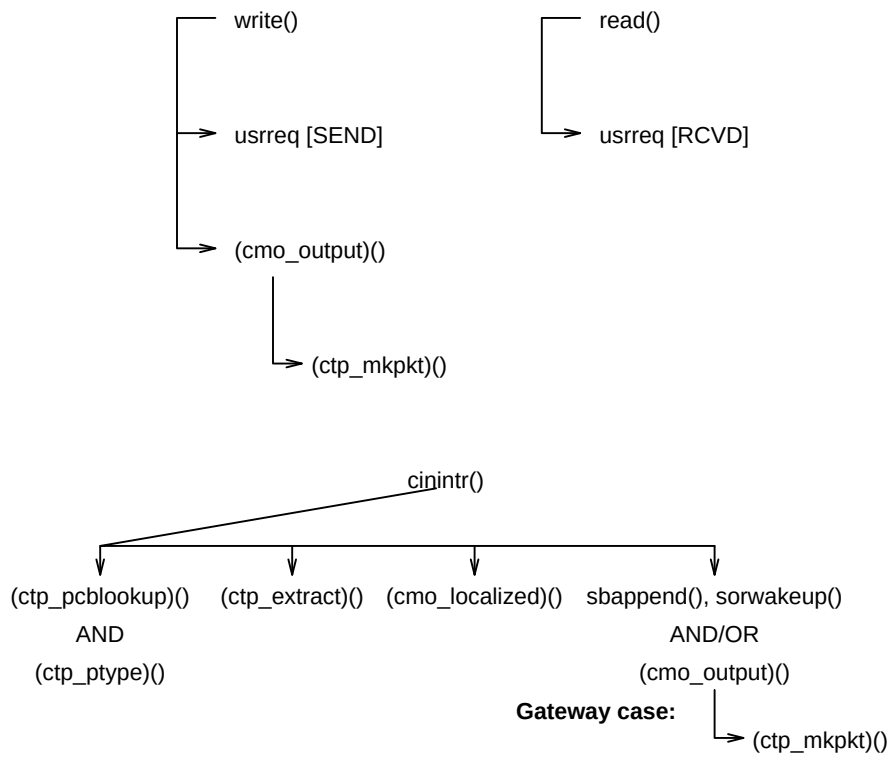



Figure 4.10: Data transfer

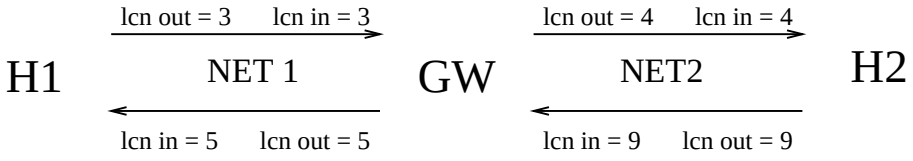



Figure 4.11: Data transfer through a gateway

control input module. This module forwards the close packet with `cmo_output()`, disconnects the socket with `cmo_discon()` and then frees the PCB. A trace of this process is shown in Figure 4.12.

## 4.3. COIP-K MODULE INTERCHANGE

### 4.3.1. MODULE INTRODUCTION

COIP-K has been created to easily design, implement, and test COIP protocols, as was shown in Figure 1.1. COIP-K is based on the idea that different COIP protocols share

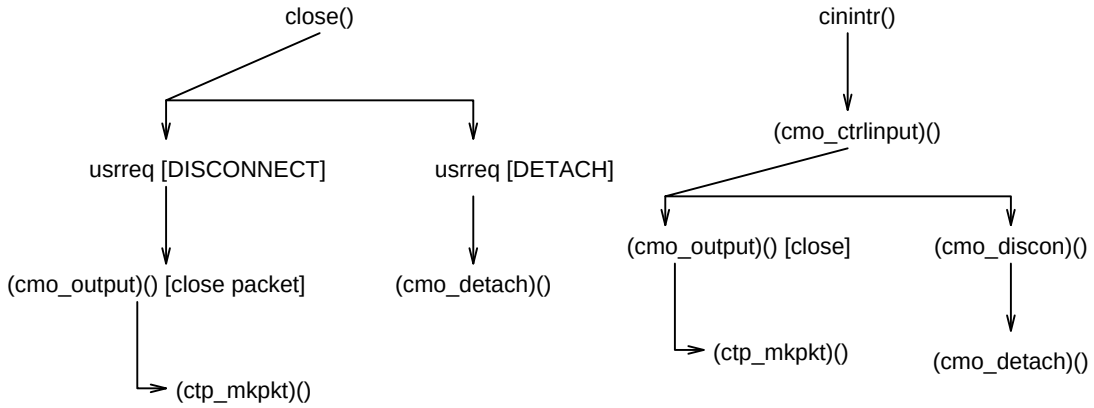



Figure 4.12: Connection shutdown

many of the same basic functions. For example, all COIP protocols, by definition, provide some sort of connection-oriented service using a simple connection state machine. To ease the COIP protocol implementation process, COIP-K implements these common functions, which consist of connection set up, data transfer, and connection termination. COIP-K then allows the more protocol-specific functions of a protocol to be plugged in as modules. Thus, the implementation of a COIP protocol with COIP-K consists of two parts: the protocol-specific functions and the common COIP-K code. The protocol-specific functions of a COIP protocol are called a COIP-K module set.

An important objective of the COIP-K package is that it allow multiple instances of COIP protocols to be installed in the kernel under it at the same time (this makes the comparison of different protocols easier). To achieve this objective, COIP-K uses the operating system's `protosw` structure in conjunction with the module set idea.

A COIP-K module set is defined by the `coip_proto` structure. This structure consists of a list of pointers to the various functions that implement the modules. Each function is its own COIP-K module. In fact, the `coip_proto` structure is very much like the system-wide `protosw` and `domain` structures, which allow multiple protocol families to use the same hardware interface without interfering with each other.

All that has to be done to create a COIP-K module set is to make a copy of the structure that defines it and set the pointers appropriately, with all the COIP-K modules that are associated with the COIP protocol implemented. The structure can be filled with two different types of modules: required modules and optional modules.

Required COIP-K modules are ones that contain the details of the COIP protocol to be implemented and are usually provided from a source external to the COIP-K package. On the other hand, optional COIP-K modules can come from sources outside the COIP-K package, or they can come from a set of COIP toolbox modules which come with COIP-K. The toolbox modules provide reasonable default modules, and they can be easily replaced by different modules, just by changing a single pointer in the module set data structure. Figure 4.13 shows how protocol modules plug into COIP-K for CTP.

By providing the COIP-K toolbox modules and allowing them to be interchanged easily with other modules, COIP-K provides a very useful feature: incremental protocol development support. When protocol developers start with COIP-K they can have COIP-K do most of the work by using many toolbox modules. As the developer becomes more advanced with COIP-K and kernel programming they can swap out toolbox modules and replace them with modules of their own. Eventually they may in fact swap out most of COIP-K and replace it with their own code. The advantage of this is that COIP-K provides as much (or as little) support as the protocol developer needs.

One of the most important ideas behind COIP-K is the idea of plugging module sets into COIP-K to easily get different COIP protocols. This process is shown in Figure 4.14. Modules can usually be shared between any number of COIP-K module sets. This saves space and provides a powerful tool for experimentation.

A programmer using COIP-K can choose which COIP-K module set he or she wants to use at socket creation time. Recall that the `socket()` system call has three arguments:

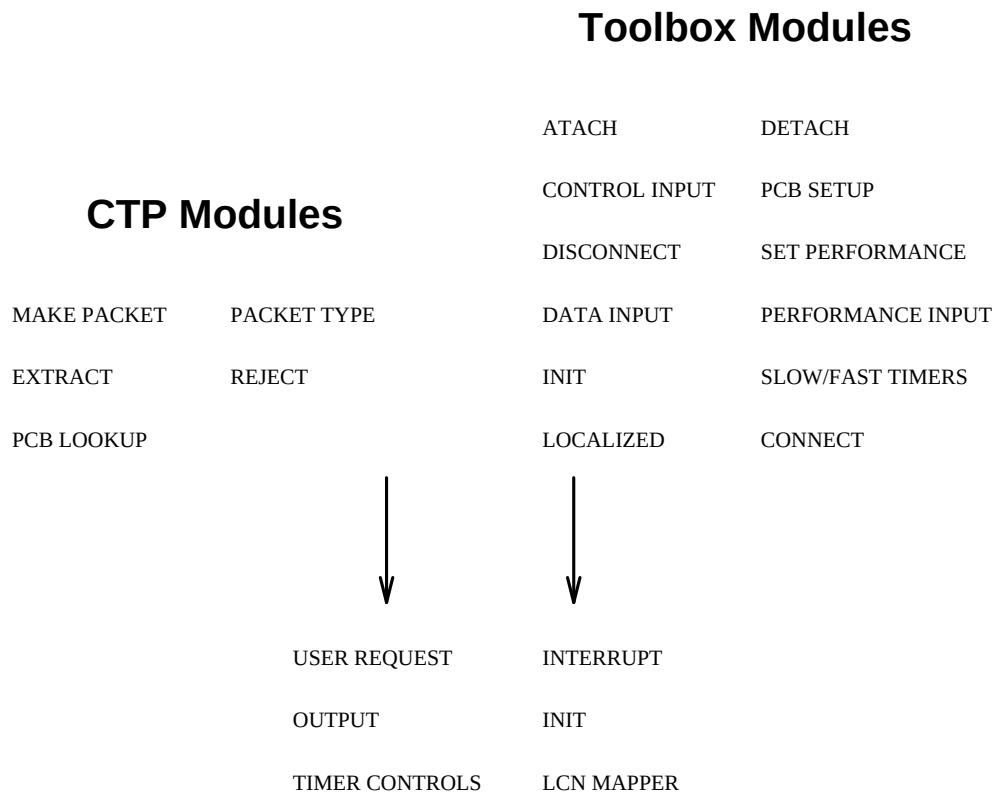



Figure 4.13: COIP-K plug-in modules

the protocol domain, the socket type, and the protocol. The third argument (the protocol) in the socket call is used to distinguish between COIP-K module sets. If the third argument is 0, the default COIP-K module set is chosen.

By providing such an easy-to-use interface at the system call level, COIP-K has made the testing and use of COIP protocols relativity easy and will hopefully minimize the number of kernel rebuilds and reboots required for testing.

Also, since COIP-K lays out the basic structure needed to make a COIP protocol (as compared to starting with just the basic `protosw` structure) it makes implementing a COIP protocol easier. This is achieved by COIP-K by dividing the tasks needed for

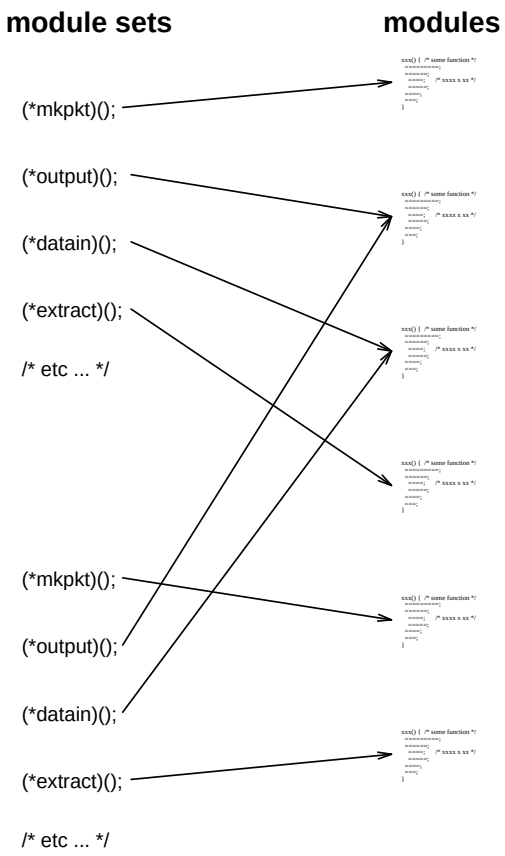



Figure 4.14: Plugging modules into COIP-K

implementing a COIP protocol up into smaller easy-to-write modules and providing example modules to guide a programmer.

### 4.3.2. MODULE SET DEMONSTRATION

In order to demonstrate the power of COIP-K in facilitating module interchange, a new COIP protocol based on CTP has been created. A few modules have been changed to modify the way multipoint connections are handled by this protocol.

There are two ways to do multipoint connections: many-to-many and one-to-many. CTP implements multipoint connections in the many-to-many fashion. This means that any data written by an endpoint on a CTP multipoint connection will be received by all

the other endpoints on the connection. Thus, the multipoint connection in this case is similar to a broadcast channel.

On the other hand, in a one-to-many multipoint connection, the connection is considered a tree, with the endpoint which established the connection at the root. When the root of the tree writes on a one-to-many multipoint connection, all the other endpoints get the message. However, when a non-root endpoint writes on the connection, only the root receives the message (although the message may pass through several gateways on the way to the root). The one-to-many multipoint connection is illustrated in Figure 4.15.

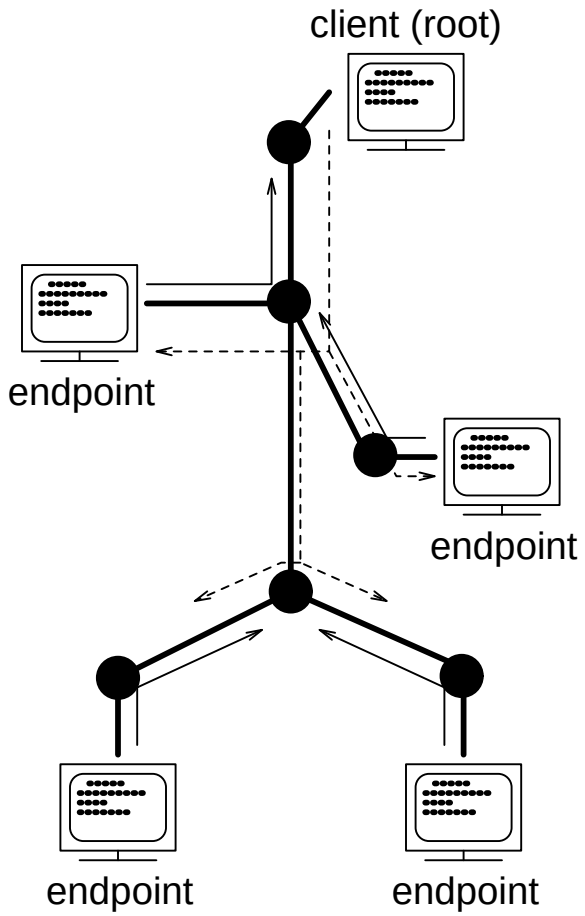



Figure 4.15: One-to-many multipoint connection

To demonstrate the COIP-K module concept, the many-to-many CTP COIP-K module set was modified to create CTP2. CTP2 is the same as CTP for point-to-point connections, but it is one-to-many for multipoint connection.

The process of creating new COIP-K protocols is very simple. The easiest way to do it is to clone an existing COIP-K module set and then modify it to create the new protocol. To clone a COIP-K protocol, all that has to be done is to edit the file `cin_proto.c`. The file contains a list of `protosw` structures and `coip_proto` structures. Except for the protocol number (which must be reassigned), the `protosw` structure can be copied. The timer functions in the `protosw` structure should be set to `NULL` (except in the CTP protocol).

Then the `coip_proto` structure can be copied. Once the `protosw` and `coip_proto` structures are copied and the system has been recompiled and rebooted, there will be two copies of the same protocol installed. At this point alternate modules can be substituted into the new module set to modify the cloned protocol. Applications need only specify the correct protocol number in the `socket()` system call to access the correct protocol.

For CTP2, the CTP module set was copied. It was then noted that the main differences between CTP and CTP2 were in packet forwarding and the time that data is passed from the protocol to the socket layer. The packet forwarding scheme of CTP2 is shown in Figure 4.15. The root endpoint should never forward a received packet, and a non-root endpoint should only forward a packet up the connection tree. Data should be passed up to the root endpoint at all times in CTP2. On the other hand, the non-root nodes should only receive data from the root node.

To handle the packet forwarding requirements of CTP2, the output module had to be modified. CTP used the COIP-K toolbox module `cmo_output` for its output module.

CTP2 took a copy of the `cmo_output` module and renamed it to `ct2_output` and modified it to forward packets as per the requirements of CTP2. Then the output module pointer in the `coip_proto` module set for CTP2 data structure was modified to point to the new function.

When the CTP protocol receives a data packet it passes the data to the socket layer, if possible, by not providing a data input module. However, for CTP2 this is not acceptable. So, a data input module was added to the CTP2 module set.

By making these two changes to the CTP module set, it was possible to create CTP2 in short order. To test CTP2, the revised simple multipoint demonstration described in Section 4.5.5 was recompiled to use CTP2, and it worked as expected.

To implement CTP2 from scratch would take a fair amount of time without COIP-K. To implement CTP2 given an implementation of CTP would not take as long as doing it from scratch, but it still would take a fair amount of time. However, with the modular nature of COIP-K, implementing CTP2 took about an hour.

### 4.3.3. ADDING MODULES TO COIP-K

Adding new features to COIP-K can be achieved by adding new modules to the `coip_proto` module set data structure. This is relativity easy to do. By adding a new pointer to a function to the end of the structure a new module can be added without having to modify existing protocol module sets which do not use the new module. When calling the new module, one should first check to see if its pointer is null or not. If it is null, then the module does not exist for that protocol. Otherwise, it can be called in the usual way.

For example, if one wanted to add support for adding an endpoint to a connection to COIP-K, the first thing to do would be to add a new module to perform that function

to the `coip_proto` structure. Then the user request function would have to be modified to catch the "add an endpoint" request (either as a `setsockopt()` or an overloaded `connect()` call) and have it call the new module, as described above.

## 4.4. COIP-K PERFORMANCE

To verify that COIP-K works properly and to quantify the performance of COIP-K, a study of CTP's performance has been undertaken.

### 4.4.1. THE COST OF COIP-K

While the COIP-K module system is useful, as shown by the CTP2 demonstration, its flexibility can not be achieved at zero cost. To determine the cost of using COIP-K to implement a COIP protocol, it is useful to compare it to the cost of implementing a COIP protocol directly (without using COIP-K). In comparing the two implementation methods, two important costs of COIP-K become apparent.

The first component of the cost of using COIP-K has to do with the added overhead of calling a COIP-K module. For example, in the performance-critical data path of COIP-K (in the interrupt function), modules are called in six places. First COIP-K must trace through the list of module sets (calling the packet type module) until a module set claims the incoming packet. The cost of this depends on how many COIP-K protocols are installed. Then COIP-K must call the protocol control block lookup module to find the PCB associated with the packet. Then the data extraction module is called twice to get the data length and data from the packet. Next, the output module is called to forward the packet along the path of the connection. Finally, the output module can call the make packet module to create a packet. This process is shown in Figure 4.16.

In a direct implementation of a protocol, function calls can be accessed directly. However, in COIP-K these calls go through the `coip_proto` data structure. This adds

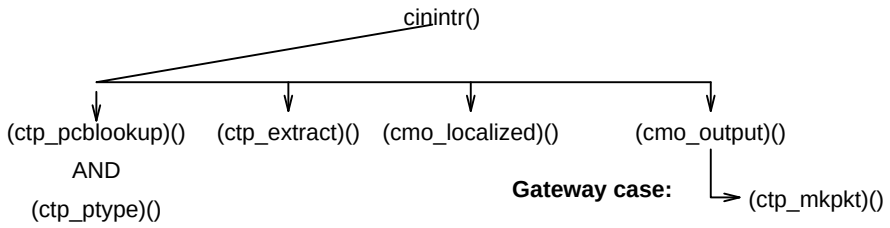



Figure 4.16: COIP-K critical path

certain indirection as shown in Table 4.1. The table shows the cost of a function call in

Table 4.1: Overhead of COIP-K module calls in Sparc machine instructions

| **Function Call Type** | **Number of Instructions** |
|---|---|
| fcall() | 1 |
| (*ptr1->fptr)() | 4 |
| (*ptr1->ptr2->fptr)() | 5 |

terms of number of instructions on a Sun Sparc1. The first row shows that it takes one instruction to call a function directly. COIP-K makes module calls as in the last two rows. If the pointer to the COIP module set is known in advance, then the cost is 4, as per the middle row. However, if the module set is not known, then it can be determined from the COIP-K PCB at the cost of another indirection, with a cost of 5, as shown in the last row of the table.

The other cost associated with using COIP-K has to do with accessing data in a packet. COIP-K directs all access to a packet's internal structure through the extract and make packet modules. If a protocol is implemented directly, the data can be accessed directly without the need for a function call. So, the added overhead here is one function call.

Another aspect of the cost of COIP-K is the code size and memory used by it. If a COIP-K module set contains $F$ functions and there are $N$ module sets active on a machine, it is possible for COIP-K to have $N * F$ functions installed in the kernel.

However, this does not take into account the fact that COIP-K modules can be reused. For example, if two protocols are exactly the same except for one module, the only extra cost is the memory needed for the modules, and the extra memory used by the `cin_proto` structure.

With today's machines, the overhead introduced by COIP-K is not substantial. For example, the `cin_intr` interrupt function has about 600 Sparc instructions in it, and only five module calls in it. Each module call costs four instructions, which is three more instructions than a normal function call. We believe that the benefits out-weigh the cost. If performance becomes an issue, one could easily modify a COIP implementation by deleting the indirections at the cost of COIP-K flexibility.

### 4.4.2. COIP-K THROUGHPUT

Protocol throughput was studied using a client and a server program. The server program simply accepts a connection and reads data until the connection is closed (the data is discarded). The client program takes two parameters: the total number of bytes to transfer, and the number of bytes in one `write()` system call (i.e. the "write size"). After the client program transfers the number of bytes specified to the server, it reports the throughput in Mbps.

The throughput performance measures were made by transferring 80MB of random data through the loopback network interface[2]. From the results of this experiment it was discovered that the theoretical bandwidth limit of the ethernet can be exceeded by UDP, COIP-K, and TCP[3]. This indicates that performance measured over the ethernet will be limited by the ethernet hardware, not the protocol processing. Thus, when measuring protocol performance it makes sense to use the loopback network so that the

[2]The loopback network interface allows a host to make a connection to itself without going over any physical network.

[3]Note that the workstation CPU is not doing anything but sending and receiving data.

limitations of the underlying network hardware do not shade the performance of the networking software.

Once the client and server programs were tested with TCP, it was easy to port them to CTP and UDP. Then, the same experiment was run for the three protocols on a Sparc1. The throughput results are shown in Figure 4.17. At a high write size, UDP performs the best, followed closely by COIP-K. TCP does not perform nearly as well due to its overhead. However, at low write sizes, TCP seems to outperform both UDP and CTP. It should be noted that comparing CTP to TCP directly is not appropriate because TCP is a transport protocol and provides flow and error control, whereas CTP is an internet protocol and does not provide any flow and error control. However, the relative performance suggests indirectly that COIP performs well.

The reason TCP performed better at low write sizes is because TCP aggregates data, while UDP and COIP do not. This was discovered by checking the read size on the server side of the connection[4].

Once it was discovered how TCP was behaving, a detailed study of the TCP documents uncovered the `TCP_NODELAY` socket option which prevents TCP from queueing up data. The results of turning on this option are shown in Figure 4.17.

It should be noted that UDP's performance over the loopback network is sometimes improved in the kernel by having it by-pass the IP and network interface layers and inputing it's data directly into `udp_input`. The data presented for UDP in this section has this optimization turned off to prevent UDP from having an unfair advantage over COIP-K.

Given that CTP performs as well as UDP, a simple and efficient implementation of a protocol by a vendor, COIP-K has been implemented efficiently.

[4]If the average read size is roughly equal to the average write size, then it is safe to assume that the size of the data in the packets is equal to the write size.

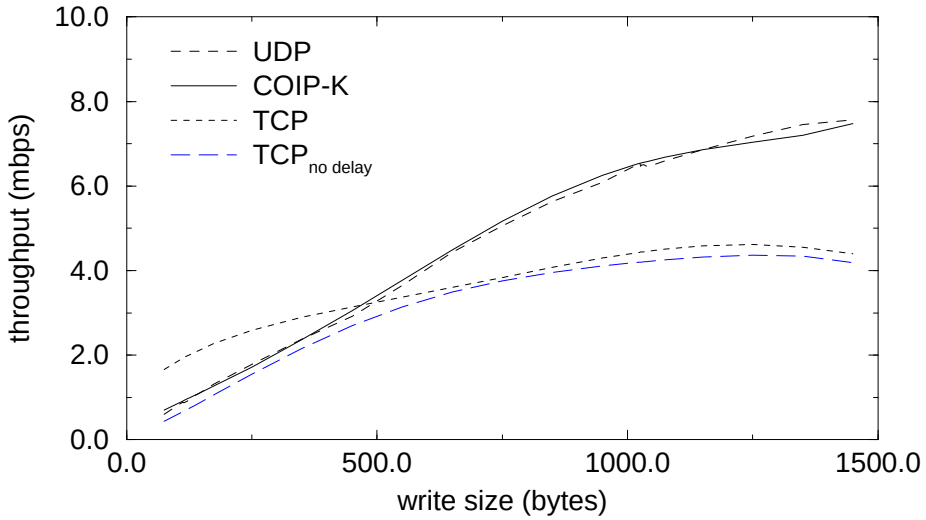



Figure 4.17: Throughput with TCP_NODELAY option added in

### 4.4.3. COIP-K DELAY PERFORMANCE

To examine COIP-K's delay characteristics, probes were inserted into COIP-K, UDP, and TCP code [10]. These probes make a timestamp at the beginning and end of the processing of a packet. Thus, by using the probes, it is possible to find out how much processing time is spent in the protocol layer of the kernel. The plots of the processing delays of COIP-K, TCP, and UDP for both client and server are shown in figures 4.18 and 4.19 respectively. The plots were generated by measuring the processing time of 200 packets, each of which contained 1024 bytes of user-data. Table 4.2 shows the minimum, maximum, and average delay for the different protocol plots.

Table 4.2: Minimum, maximum, and average protocol delays in msecs.

| Protocol | Minimum | | Maximum | | Average | |
|---|---|---|---|---|---|---|
| | client | server | client | server | client | server |
| TCP | 0.383 | 0.209 | 9.641 | 1.284 | 1.796 | 0.251 |
| UDP | 0.082 | 0.064 | 1.565 | 0.113 | 0.146 | 0.073 |
| COIP-K | 0.071 | 0.043 | 1.161 | 0.079 | 0.115 | 0.048 |

By taking the average time spent in COIP-K processing from the table, it is possible to calculate the maximum throughput COIP-K could generate on a Sparc1 if it only had to do protocol layer processing. It was found that CTP spent 48 $\mu$secs receiving data and 115 $\mu$secs sending data (with a packet size of 1024 bytes). By taking the inverse of the larger of these two times, one gets the number of packets that can be processed in one second giving a theoretical maximum throughput of 71.235 Mbps. All this data suggests that the per-packet processing of COIP-K has been implemented efficiently.

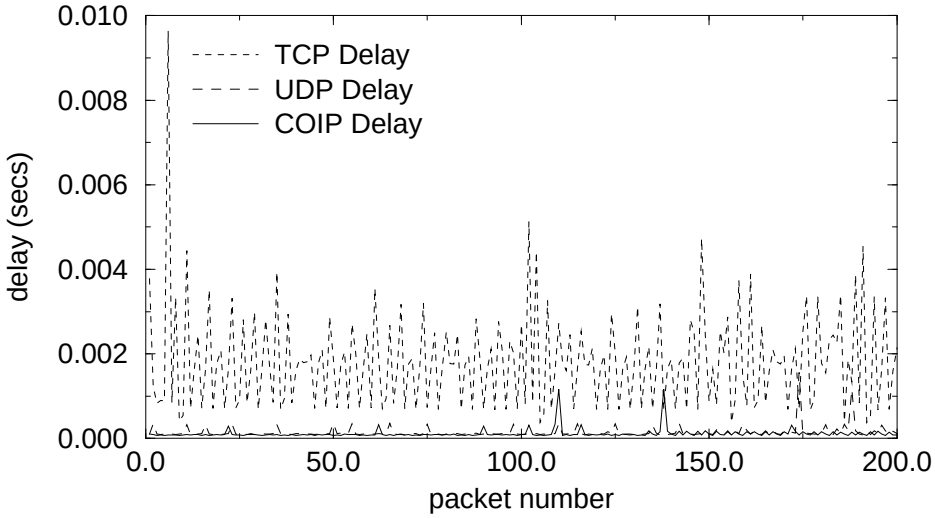



Figure 4.18: Client delay plot

### 4.4.4. COIP-K'S EFFECT ON QUEUE LENGTH

Queueing occurs in two main places in the kernel. For outbound traffic, the network device interface queue stores packets until the ethernet hardware is ready to send a packet. For inbound traffic, the receiver's socket buffer is used to store data until the receiver reads the data. The probes used to measure delay were designed to be used on traffic over the ethernet. It was found that on a Sun Sparc1, both UDP and COIP-K were

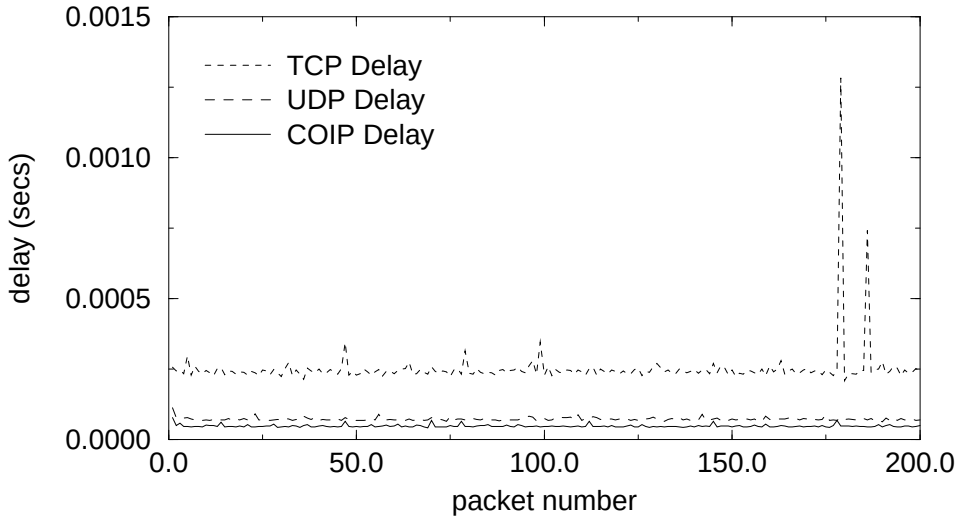



Figure 4.19: Server delay plot

able to overflow the outbound network interface queue. This indicates that COIP-K can operate faster than ethernet speeds.

On the loopback network it was found that COIP-K overflows the receive socket queue, which means that the COIP-K was generating data faster than the receiving process could read it.

## 4.5. COIP-K TESTING AND DEMONSTRATIONS

The eight test programs run using CTP are presented in the next subsections. The same test programs can exercise different parts of the COIP-K code depending on what hosts are members of the connection. Figure 4.20 shows the three main testing configurations: loopback, over a local network, and through a gateway. All applications have been tested in these configurations.

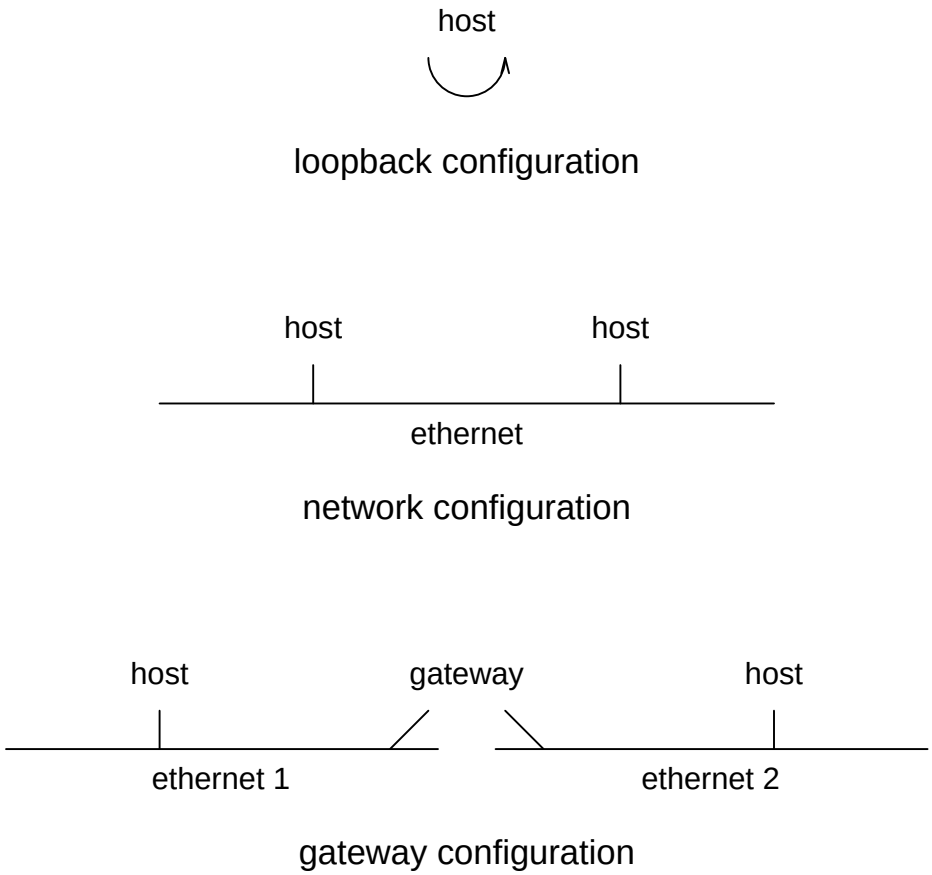



Figure 4.20: COIP-K testing configurations

### 4.5.1. SIMPLE DEMONSTRATION

The first set of programs written to test COIP-K connections were simple point-to-point client and server programs. These programs were intended to be a first test of the point-to-point aspects of COIP-K. They can be run over a local network, or through a host with multiple network connections (to test the gatewaying portion of the code).

To run the simple demonstration, the server is first started. Then, the client is started and makes a COIP connection to the server. Then the client sends one data packet to the server. The server receives the packet and sends a data packet back. Then the connection is closed. An attractive aspect of the client and server programs

is that they are interactive and execute COIP-K system calls only when the operator indicates they should. This means that each program does one step in the connection process (i.e. one system call) and then pauses. This is very useful because there are only a few COIP events involved, and they happen slowly. This makes it easy to pinpoint problems in the code.

### 4.5.2. FILE TRANSFER TEST

The next set of test programs written for COIP-K implemented a go-back-n sliding window protocol in user code on top of CTP. The test programs used the sliding window protocol to transfer a file from the client to the server over a CTP point-to-point connection. The programs test the point to point code and show that COIP-K/CTP can have a transport protocol built on top of them to perform a useful task (in this case, a file transfer). The set up for this demonstration is shown in Figure 4.21.

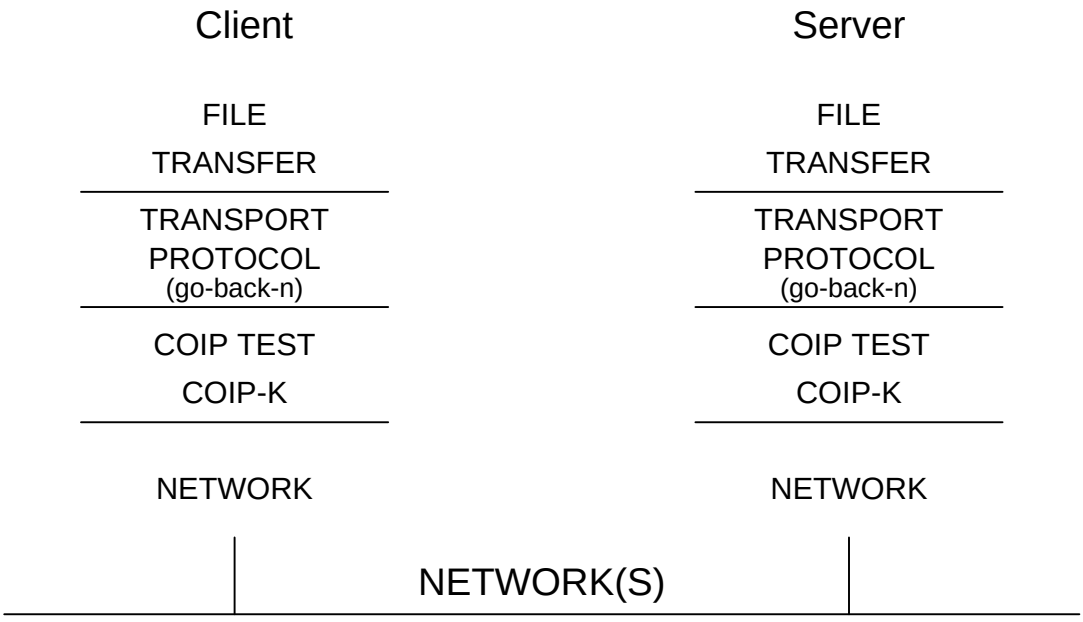



Figure 4.21: File transfer program

This test is more complex than the simple demonstration because it generates a lot more COIP events in a short period of time. It can be run over a local network, or through a COIP-K gateway.

### 4.5.3. FILE TRANSFER THROUGH "GATEWAY"

This test is similar to the previous test, except instead of the client establishing a connection directly with the server, it establishes the connection through a third server (called `xserv`). The `xserv` program runs as a user process and acts as a user level "gateway" between the client and the server. Depending on what mode `xserv` is run in, it can act as a reliable gateway, or it can drop or reorder packets with certain probability which is specified by the user. This test is shown in Figure 4.22.

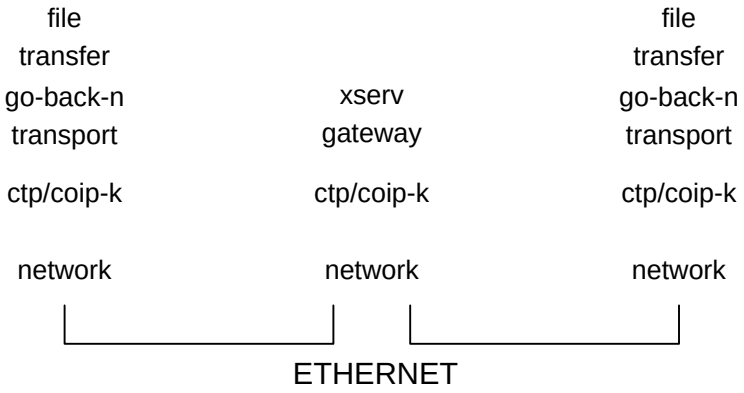



Figure 4.22: File transfer program, through a third party

The `xserv` host accepts a COIP connection from the client and establishes another COIP connection to the server. Once the connection is established, the client process will generate data as fast as possible. The `xserv` process must accept this data and forward it to the server with variable reliability as specified by the user. The transport protocol will recover from the errors introduced by `xserv`. A successful run of this test shows that COIP-K can handle a large number of data requests properly and that it also can handle multiple CTP connections at once. This test is the most complex test of the

point-to-point tests. The bulk of the complexity is at the host which runs the `xserv` program. It has to deal with multiple COIP connections and a large number of COIP packets per second.

### 4.5.4. TELNET DEMONSTRATION

The final COIP point-to-point demonstration programs were a port of the 4.3 BSD Unix telnet/telnetd programs. This allows a COIP-K program to be used for remote login. While this test is not as complex as the previous one, it works and has run for several hours without a connection breakdown. It took about ten minutes to modify telnet to run with CTP. This test shows that existing applications such as telnet can be ported to a COIP protocol without any difficulty.

### 4.5.5. SIMPLE MULTIPOINT DEMONSTRATION PROGRAM

In the simple multipoint demonstration program two servers are started and the client establishes a connection between the three endpoints. All three processes then enter a similar state. In this state each process is in a `read()` system call waiting for data on the connection. In addition to being in the read call, each process had also trapped the Unix `SIGINT` or interrupt signal. When this signal is received, the program interrupts the `read()` system call and enters the signal handler. In the signal handler the process sends a message out on the COIP connection and then return back to the `read()` system call.

The result of this set up is shown in Figure 4.23. Once the connection is established, if the operator hits control-C on one of the processes, it sends a message out on the connection to the other two processes. The other two processes then display the message.

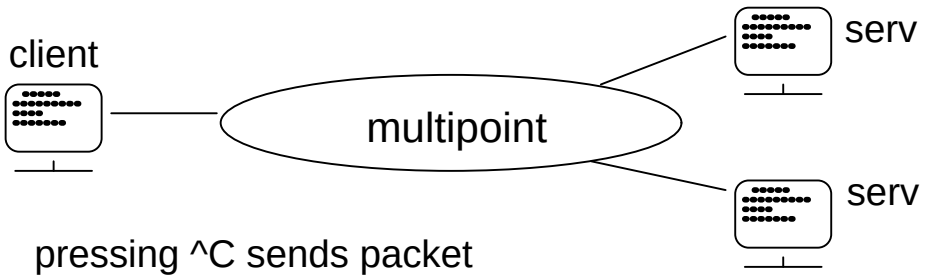



Figure 4.23: Revised multipoint test

This test was of great use in debugging the multipoint code in COIP-K and CTP (especially the connection setup routines). It also served as the stepping stone in developing the next two applications.

### 4.5.6. MULTIPOINT CHAT PROGRAM

Another demonstration of COIP-K multipoint connections was the COIP multipoint chat program. In this program a multipoint connection is established between three processes. The program then divides the screen into two parts: an input region and a display region, as shown in Figure 4.24. Anything a user types is displayed in the input

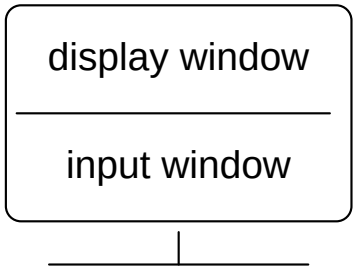



Figure 4.24: Multipoint chat program screen layout

region until the user hits return. Then the typing is transmitted over a CTP connection to the display part of all the other machines.

This sort of program is the simplest case of a multiparticipant collaboration. Thus, it shows that such applications can be done on top of COIP-K.

### 4.5.7. MULTIPOINT SCRIPT DEMONSTRATION

The final test program for COIP-K is the multipoint script program. The "script" program is a program under Unix which logs a session to a file. The multipoint script program logs a session to a COIP multipoint connection so that multiple people can watch someone's session. This process is shown in Figure 4.25.

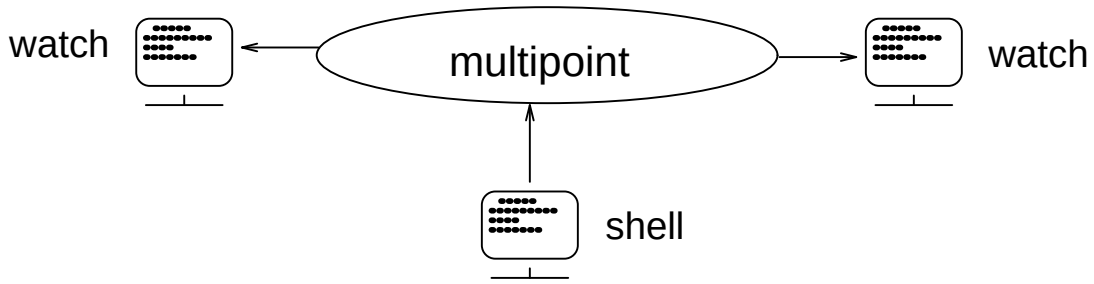



Figure 4.25: Multipoint script program

This sort of application is useful for multiparticipant collaboration. It could also be useful for distributing useful information, such as network status, to hosts throughout a network.

### 4.5.8. COIP-K NETWORK STATUS PROGRAM

The COIP-K network status program, `cinst`, is more of a debugging aid than a test or demonstration program. However, it is useful in testing the system and thus merits mention here. The `cinst` program basically reads the Unix kernel's memory and dumps out the list of active COIP-K protocol control blocks and the various state information stored in them. (This is very much like the `netstat` program which dumps out the state of all TCP connections.) This process is illustrated in Figure 4.26.

When running a test that fails in some way, the `cinst` program can be useful in determining what is happening inside the kernel.

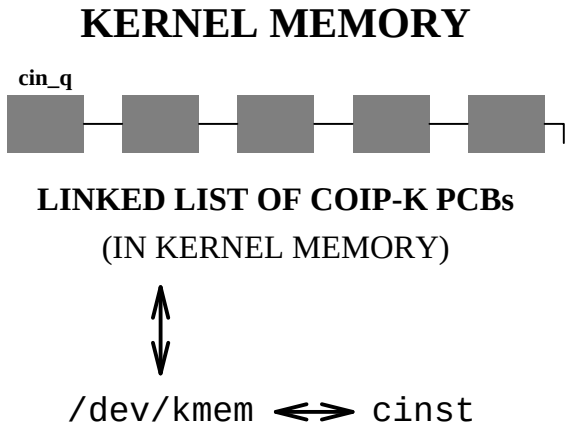



Figure 4.26: The `cinst` program

# 5. CONCLUSIONS AND FUTURE RESEARCH

## 5.1. CONCLUSIONS

Over the past few years, a number of research groups have proposed connection-oriented internet protocols (COIPs) to provide variable grades of service with performance guarantees over an internet of heterogeneous networks. Two major advantages of connection-oriented protocols include the ability to do statistical reservations for each connection and simplified per-packet processing. The proposed COIPs have a number of similarities and differences. The research groups believe that it is important to pursue these protocols and compare and contrast alternate approaches to these protocols. However, implementation of these protocols completely independently was considered unwise for two reasons. First, as the proposed COIP protocols have a number of common functions, independent implementations would lead to a lot of duplicate work. Secondly, implementation of a protocol in the Unix kernel poses a number of challenges.

In order to develop a more productive research environment, avoid duplication of work, and foster collaboration, we proposed the COIP-kernel (COIP-K). COIP-K forms the core of a COIP protocol and includes the minimum functionality necessary for a wide range of multicast connection-oriented protocols. It also includes appropriate provisions to interface other functional modules. COIP-K, when combined with a set of functional modules, will create an instance of a COIP such as MCHIP or ST.

This approach to protocol development yields important benefits to the research effort. Many of the functions that a COIP must provide can be supported by a number of

alternative mechanisms. These mechanisms can be implemented in experimental modules and integrated with COIP-K to produce alternate instantiations of COIPs. These instantiations represent different mechanisms which can be compared under controlled experimental conditions. As a result, it will be possible to describe under what conditions each of the mechanisms behaves well or poorly, and thus define a COIP that is optimal for a given target environment.

This thesis deals with the challenge of realizing the COIP-K vision and demonstrates its feasibility and viability. There were five main implementation requirements set for COIP-K before it was designed. Our implementation of COIP-K has met all these requirements as outlined in the following paragraphs.

First of all, COIP-K has been successfully implemented in the Unix kernel within the framework of the Unix networking model.

Secondly, COIP-K allows implementation of various COIP protocols by module interchange, and it supports incremental software support for COIP-K programmers. This allows easy development and evaluation of the tradeoffs associated with different COIPs. To demonstrate the flexibility of the module interchange concept, we used the COIP Test Protocol (CTP), which treats multipoint connections as a broadcast channel among endpoints and allows many-to-many communication. Using module interchange, a new instance of CTP (called CTP2) was created with modified multipoint behavior to allow one-to-many instead of many-to-many communication.

Our implementation of COIP-K has retained most of the user-level socket interface. This allows existing applications to be ported to COIP-K without any difficulty. To demonstrate this, we successfully ported standard 4.3 BSD `telnet` to work on top of CTP.

The COIP-K code has efficient per-packet processing. We found that COIP-K's throughput and delay performance are comparable to UDP, and are much better than those of TCP. Also, we found that COIP-K could easily overwhelm the ethernet hardware interface on Sun Sparc stations. Based on the processing latency of COIP-K on a Sparc1, the theoretical maximum data rate of COIP-K can be as high as 86.2 Mbps.

Finally, COIP-K supports multipoint connections. This is useful for a variety of applications, including multiparticipant collaboration applications. We developed two basic multiparticipant collaboration applications to demonstrate this.

## 5.2. FUTURE RESEARCH

The research work in this thesis can be extended in several ways. At the highest level, CTP could be enhanced until it becomes a full implementation of MCHIP. This would involve creating appropriate resource allocation and enforcement modules for COIP-K.

Another possible extension to COIP-K is to allow for more extensive connection management options such as the addition and deletion of an endpoint from a connection. This functionality is important for a number of multipoint applications.

In the future, COIP-K could be modified to support more than one addressing scheme. The current assumption of IP addressing was used mainly to avoid address resolution on the ethernet.

COIP-K could also be extended by porting it to the NeXT platform. Then it could be interfaced to the NeXT ATMizer which has been developed at Washington University.

Another possible extension to the COIP-K research is to develop application-oriented light weight transport protocols (ALTPs). These ALTPs could run on top of a COIP-K based protocol and be used to support real applications.

Finally, module sets for other COIP protocols such as ST and FLOW could be developed for COIP-K. This would allow interesting comparisons of the different COIPs under a variety of scenarios.

# 6. ACKNOWLEDGMENTS

First, I would like to thank my advisor, Guru Parulkar, for his support and friendship throughout my college career. Second, I would like to thank Jon Turner, Rick Bubenik, and Martin Dubetz (my committee) for reading my thesis and providing useful comments and suggestions to improve it. Third, I would like to thank Lorrie Faith Ackerman (my favorite editor) for proof reading countless versions of the thesis.

In addition to the people directly involved in the thesis process, I would like to thank the people in St. Louis who have kept my social life afloat the past few years. First, I would like to thank Gaurav Garg and Sarang Joshi for making me feel welcome during my first year of grad school. Second, I would like to thank the members of my research group G-TROUP for being there to share in the experience. In particular, I'd like to thank Zubin Dittia, Milind Buddhikot, Larry Gong, and Christos Papadopoulos for always having time to take a break and talk. Next, I would like to thank Greg Peterson and the Phools for all the great movies and for bucking the anti-social trend so prevalent in grad school. Also, I would like to thank R. Gopalakrishnan for all the great quotes.

I would also like to thank my friends from University of Delaware (especially members of the F-TROUP research group). Erik Perkins and Pam Elliott deserve special note for keeping me up to date on happenings at UDel. Additionally, John Price (Raistlin) from TAMU has my thanks for keeping me entertained over the summer of 1991.

Finally, I would like to thank my family for always being there when I needed them.

# A. BIBLIOGRAPHY

[1] American National Standards Institute Inc., ANSI X3.139-1987 *Fiber Distributed Data Interface (FDDI), Token Ring Media Access Control (MAC).*

[2] Cidon, I., Gopal, I., Kaplan M., and Kutten, S., "Distributed Control for PARIS," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pp. 145-160, 1990.

[3] Cidon, I., Gopal, I., and Guérin, R., "Bandwidth Management and Congestion Control for PARIS," *IEEE Communications Magazine*, vol 29, no 10, pp. 54-63, October 1991.

[4] Comer, Douglas, *Internetworking with TCP/IP*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[5] Dhas, Chris, Konangi, Vijay, and Sreetharan, M., "Broadband Switching Architectures, Protocols, Design, and Analysis," *IEEE Computer Society Press*, Los Alamitos, California, 1991.

[6] Forgie, J., "ST - A Proposed Internet Stream Protocol," IEN 119, MIT Lincoln Laboratory, 7 September 1979.

[7] Kapoor, S., *Design and Analysis of a Two Port* ATM-FDDI *Gateway*, M.S. thesis, Department of Electrical Engineering, Sever Institute of Technology, Washington University, St. Louis, Missouri, December 1991.

[8] Leffler, Samuel J., McKusick, Marshall K., Karels, Michael J., and Quarterman, John S., *The Design and Implementation of the 4.3* BSD *Unix Operating System*, Addison-Wesley Publishing Company, Inc., Redding, Massachusetts, 1989.

[9] Mazraani, Tony Y., and Parulkar, G., "Specification of a Multipoint Congram-Oriented High Performance Internet Protocol," INFOCOM'90, *IEEE Computer Society*, Washington D.C., June 1990.

[10] Papadopoulos, Christos, *Remote Visualization on a Campus Network*, M.S. thesis, Department of Computer Science, Sever Institute of Technology, Washington University, St. Louis, Missouri, Spring 1992.

[11] Parulkar, Gurudatta M., "The Next Generation of Internetworking," *ACM SIGCOMM Computer Communcations Review*, vol 20, no 1, pp. 18-43, Jan. 1990.

[12] Topolcic, C., "Experimental Internet Stream Protocol: Version 2 (ST-II)," RFC-1190, October 1990.

[13] Zhang, Lixia, *A New Architecture for Packet Switching Network Protocols*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, July 1989.

[14] Zhang, Lixia, "Virtual Clock: A New Traffic Control Algorithm for Packet Switching Networks," *ACM Transactions on Computer Systems*, vol 9, no 2, pp. 101-124, May 1991.

# B. VITA

Bibliographical items on the author of the thesis, Charles D. Cranor.

- Born June 14, 1967 in Camden, South Carolina.
- Attended the University of Delaware from September 1985 to May 1989.
  - Worked as a undergraduate research assistant in Professor David J. Farber's research group (F-TROUP) from January 1986 to September 1987.
  - Worked as a system manager for the EE/CIS Computer Lab, September 1987 to June 1989.
- Attended Washington University from September 1989 to present. Received Research Assistantship in the Computer and Communications Research Center. Accepted in the CS PhD program, Fall 1991.
- Member of the IEEE, and ACM.

May, 1992