Report Number: WUCS-92-16

1992

An Implementation Model for Connection-Oriented Internet Protocols

Charles D. Cranor and Gurudatta M. Parulkar

Recently a number of research groups have proposed connection-oriented access protocols that can provide a variable grade of service with performance guarantees on top of diverse networks. These connection-oriented internet protocols (COIPs) have different performance trade-offs. The purpose of this paper is to create a COIP-Kernel which can be used as a toolkit to implement any of the proposed COIPs. COIP-K features module interchanges and incremental software support. The paper presents the COIP-K implementation and its performance characterization.

AN IMPLEMENTATION MODEL FOR CONNECTION-ORIENTED INTERNET PROTOCOLS

Charles D. Cranor
chuck@maria.wustl.edu
+1 314 935 4203

Gurudatta M. Parulkar
guru@flora.wustl.edu
+1 314 935 4621

WUCS-92-16

May 5, 1992

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Recently a number of research groups have proposed connection-oriented access protocols that can provide a variable grade of service with performance guarantees on top of diverse networks. These connection-oriented internet protocols (COIPs) have different performance trade-offs. The purpose of this paper is to create a COIP-Kernel which can be used as a toolkit to implement any of the proposed COIPs. COIP-K features module interchange and incremental software support. The paper presents the COIP-K implementation and its performance characterization.

AN IMPLEMENTATION MODEL FOR CONNECTION-ORIENTED INTERNET PROTOCOLS

Charles D. Cranor
chuck@maria.wustl.edu
+1 314 935 4203

Gurudatta M. Parulkar
guru@flora.wustl.edu
+1 314 935 4621

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 [14].

Second, most of the emerging high speed networks, such as ATM [7] 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 [12], ST [8, 17], and FLOW [18, 19]. 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 [9, 5].

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.

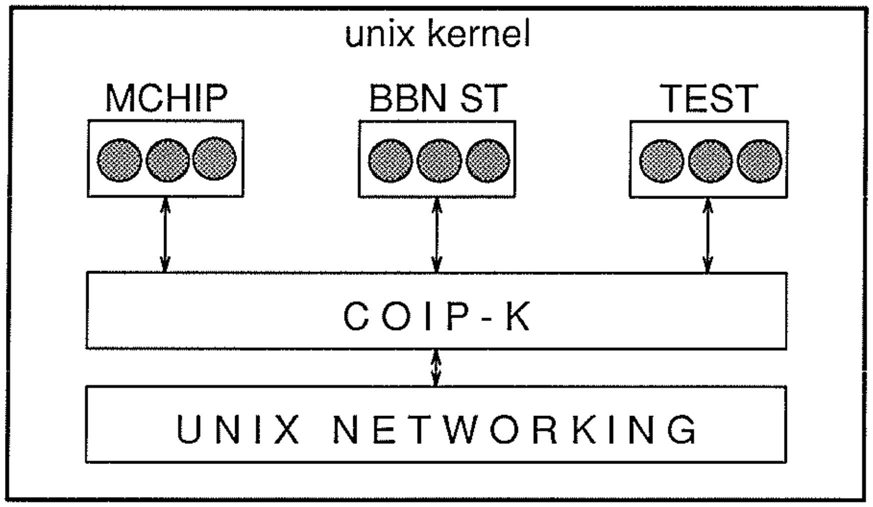


Figure 1: COIP-K structure

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. 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 support 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.

This work is somewhat related to the work done at University of Arizona on the x-Kernel [15]. However, where as the x-Kernel is a general solution to protocol implementation (with emphasis on protocol stacking and uniform interfaces), COIP-K focuses specifically on fitting COIPs in the existing Unix kernel. In fact, the latest version of the x-Kernel no longer runs inside the Unix kernel. Instead it uses raw sockets to access the network.

The topic of this paper is the challenge of realizing the COIP-K vision and demonstrating its feasibility and viability. The goals of this paper 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. Paper Outline

Section 2 presents background information and necessary details of the 4.3 BSD Unix operating system's networking system. The details of the 4.3 BSD system are presented in [11].

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

Section 4 discusses the feasibility and viability of COIP-K. The specification and implementation of the COIP Test Protocol (CTP) is first presented (code traces of CTP can be found in [6]). 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, Section 5 summarizes the contributions made by this research and discusses directions for future research.

2. Background

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.

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 really system calls which cause the system to enter kernel mode and call down to the lower networking layers. The middle layer, called the protocol layer, consists of a number of protocol suits 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 shows how the three layers are organized with respect to the user and the network. In the example at the bottom of the figure, 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.

One other important aspect of the 4.3BSD Unix networking system is the dynamic memory allocation system. This system is called the **mbuf** system (mbuf stands for memory buffer). The details of the inner workings of the mbuf system can be found in [6, 11].

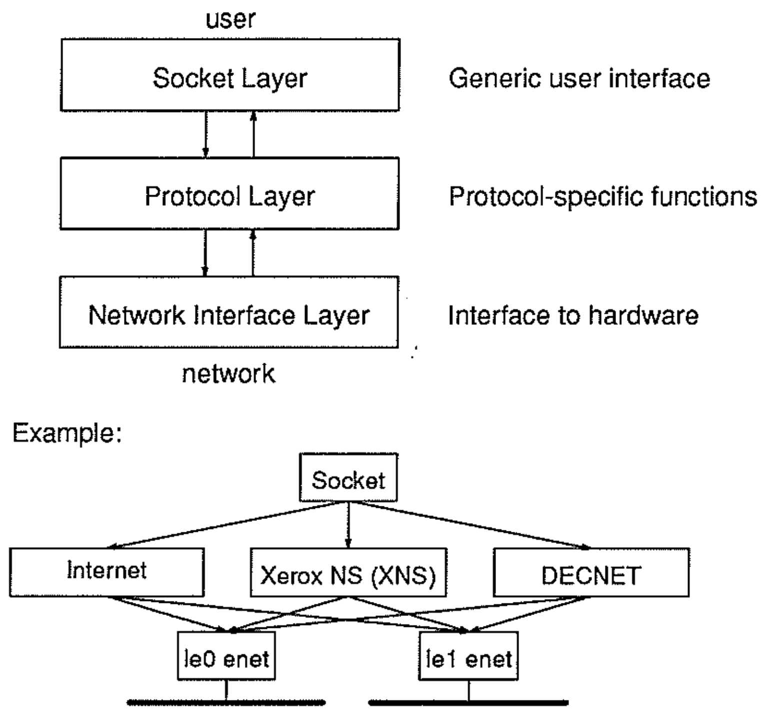


Figure 2: BSD Unix network layering

3. COIP-K Implementation

This section 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.

3.1. 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.

3.1.1. 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 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: COIP-K socket address structure

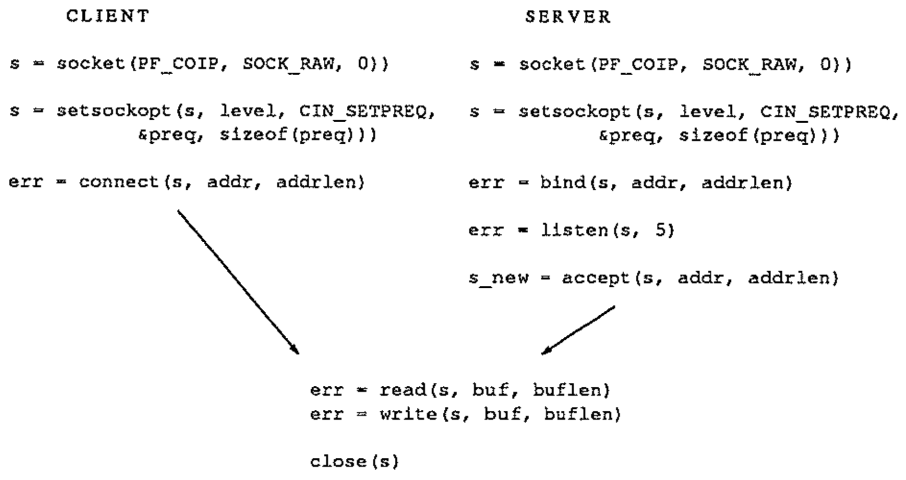
For a multipoint connection, the setup of the sockaddr_cin is more complex since the size of the socket address depends on the number of hosts in the multipoint connection. The format of the sockaddr_cin structure does not change, but there can be a number of cinmad structures appended to it. Given a variable number of hosts in a multipoint connection, it is best to dynamically allocate space for the addresses using malloc(). Figure 4 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.

3.1.2. Client Setup. The COIP-K client starts by creating a COIP-K socket as shown in Figure 5. 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 described in the previous subsection. This is followed by a call to the connect() function. If the connect() is successful,

```
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 4: COIP-K multipoint socket address structure



```
CLIENT                                   SERVER

s = socket(PF_COIP, SOCK_RAW, 0))        s = socket(PF_COIP, SOCK_RAW, 0))

s = setsockopt(s, level, CIN_SETPREQ,    s = setsockopt(s, level, CIN_SETPREQ,
        &preq, sizeof(preq)))                    &preq, sizeof(preq)))

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

                                         err = listen(s, 5)

                                         s_new = accept(s, addr, addrlen)

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

                     close(s)
```

Figure 5: Sample COIP-K point-to-point client and server

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.

3.1.3. 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.

3.1.4. 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.

3.2. 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 6. 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 6 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. 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* 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.

3.2.1. 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 7.

*Note that the "connection" is not a reliable connection. Instead, it is a cross between a reliable connection and a datagram, thus it is sometimes called a "congram."

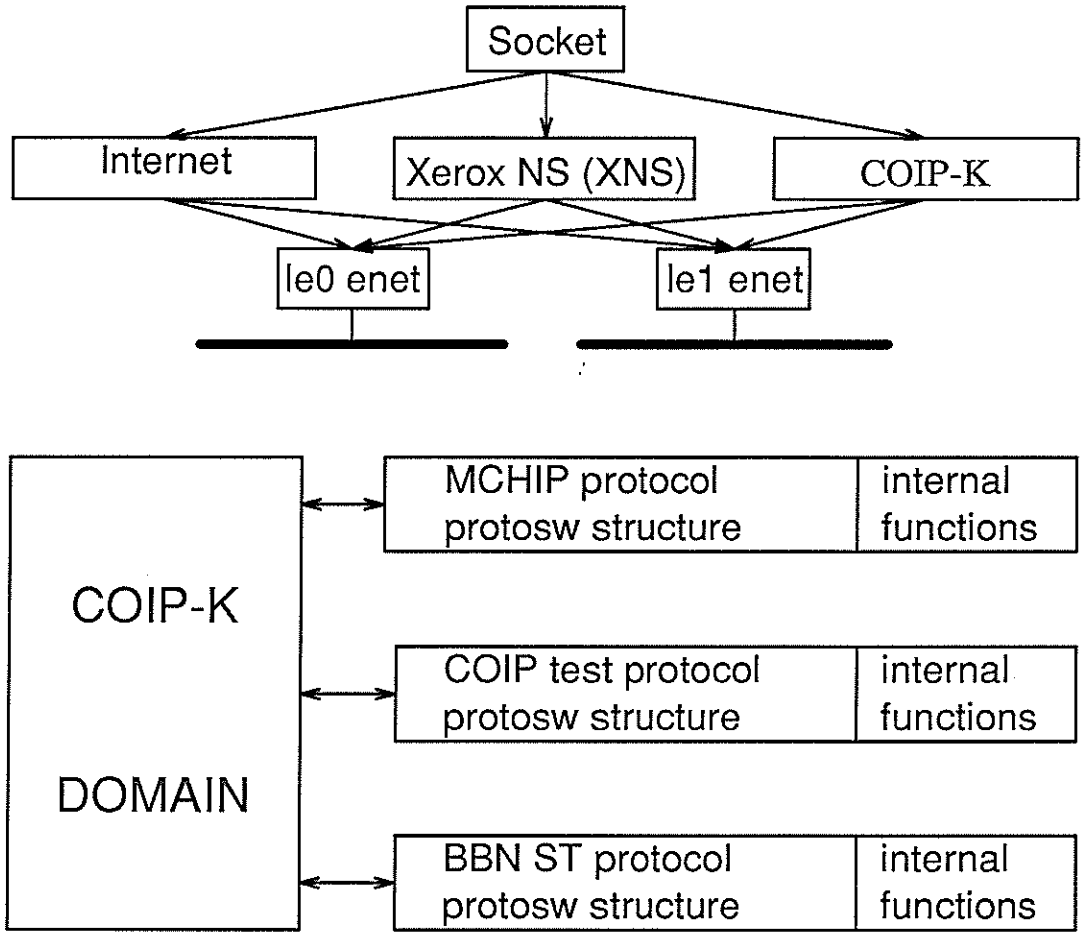


Figure 6: The BSD Unix networking model and COIP-K

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.

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, a a protocol-specific module will be called every time a COIP-K PCB is made or freed.

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

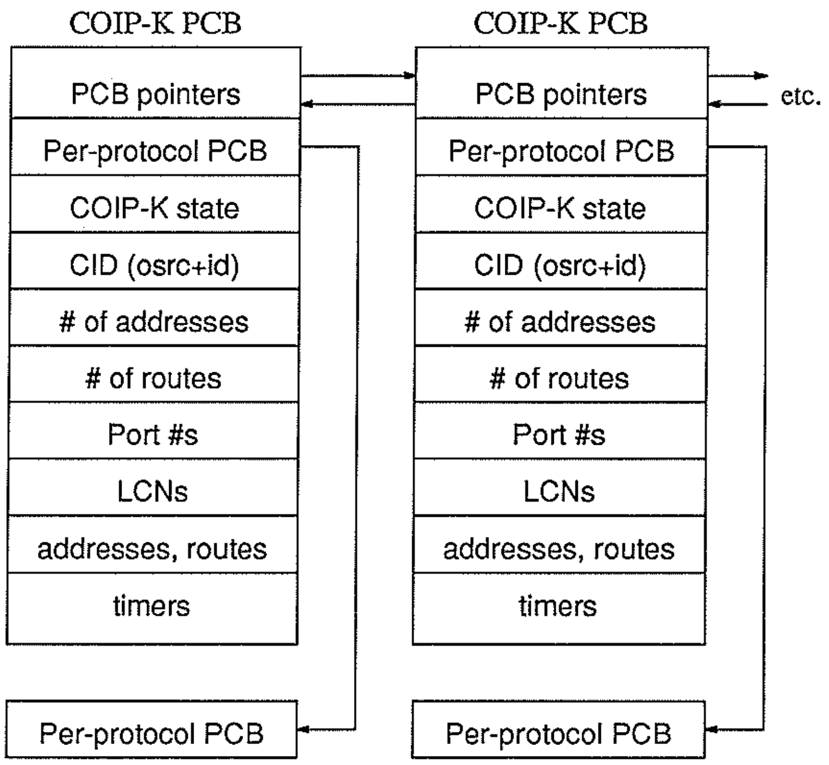


Figure 7: COIP-K PCB Structure

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 8. 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.

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

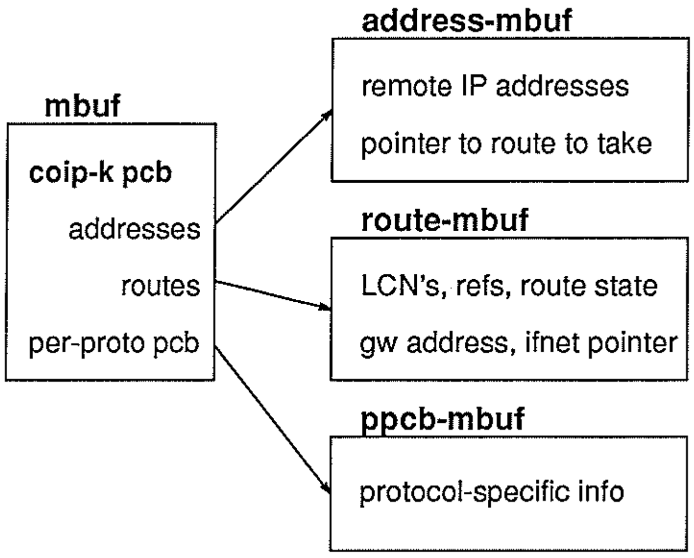


Figure 8: Multipoint PCB structure

- 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.

3.2.2. 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 9 shows how a COIP-K protocol fits in with the other layers. Note that the actual COIP-K implementation consists of a large number of functions which (for the sake of clarity) are not shown in Figure 9. The functions labeled "extract" and "mkpkt" are actually required protocol-specific COIP-K modules and are described in Section 3.3.1.

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 shown in Figure 3.2.2.

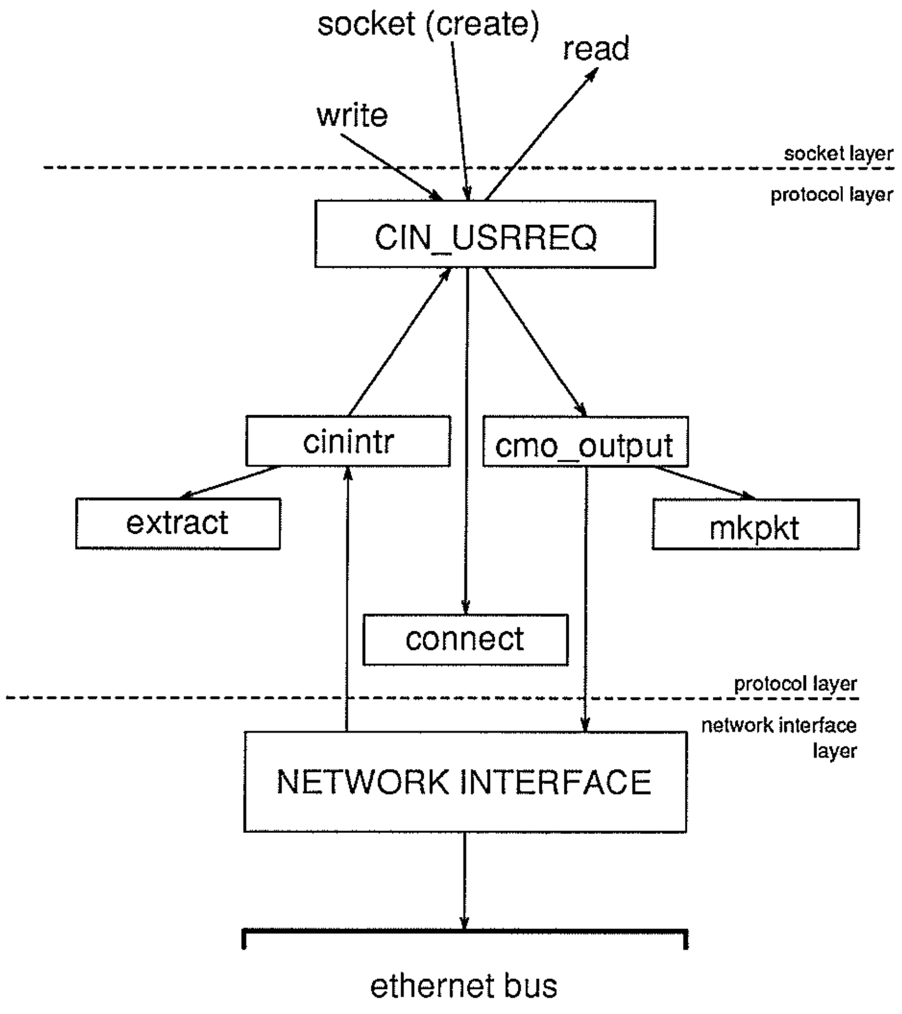


Figure 9: COIP-K protocol switch

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 shown in Figure 3.2.2.

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

```
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;
}
```

Figure 10: The cin_usreq() function

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 12. 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.

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.

3.3. 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 13. By providing default toolbox modules, COIP-K provides an incremental level of support

```
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;
}
```

Figure 11: The `cinintr` function

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 required and optional modules that make up a COIP-K protocol are called a COIP-K module set. The next two subsections list the COIP-K modules that make up a COIP-K module set.

3.3.1. 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 a packet, it uses the extract

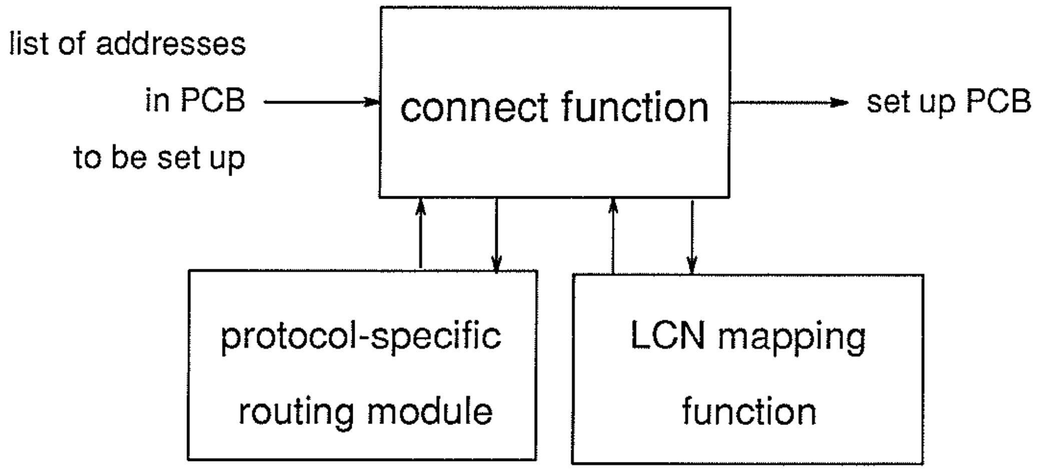


Figure 12: Connect function

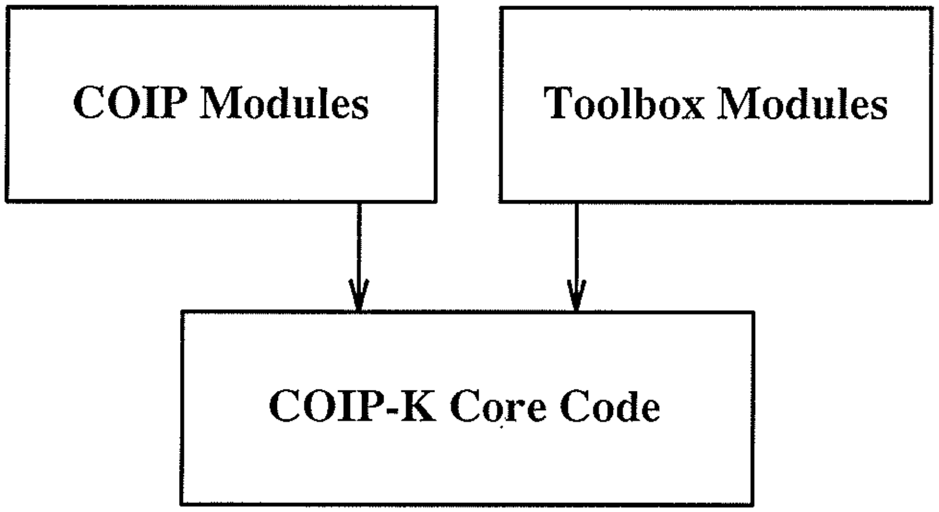


Figure 13: COIP-K module plug in

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 set of modules, 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.

3.3.2. 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 3.2.2.

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

This section 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 to include multiple module sets and incremental support.
- 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 section is to demonstrate how these objectives have been successfully achieved, and in some cases surpassed. The outline of the section follows.

Section 4.1 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 4.2 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 4.3 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 4.4 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.

4.1. 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.

4.1.1. 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 14. The open packet

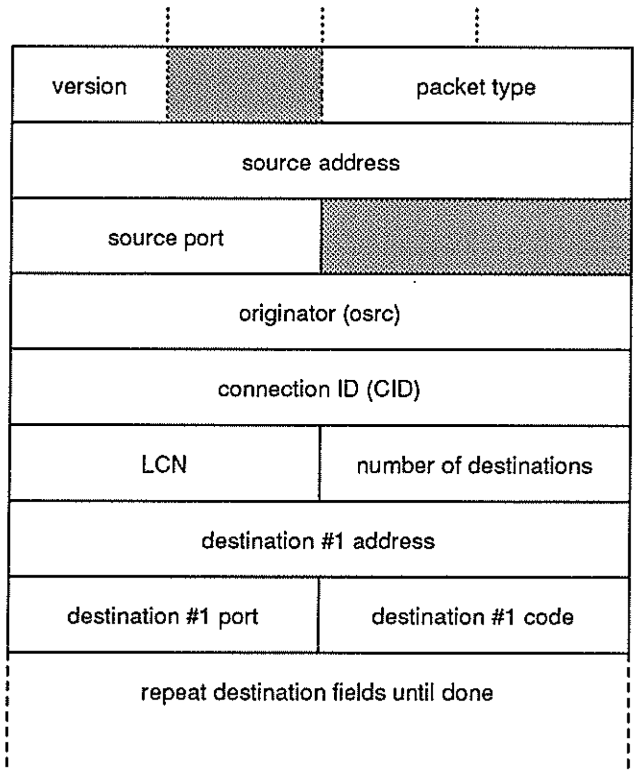


Figure 14: Open packet

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[†]. (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

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

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 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 14. 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 15 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 16 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.

CTP Data Packet. The format for a data packet is rather simple and is shown in Figure 17. In this example the packet type is set to "data." The only other information 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 18. In addition to the standard 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.

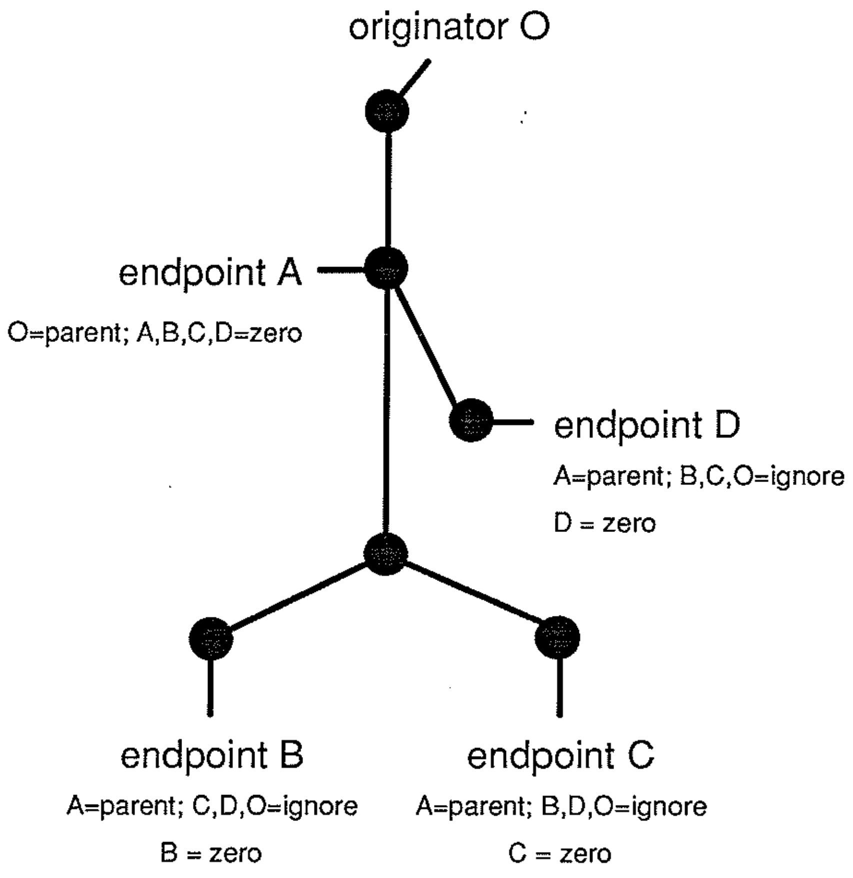


Figure 15: Example of the use of the code field in an open packet

4.1.2. 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 19. The COIP-K states are shown on the sides of the vertical time 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 20. The packets are all in the CTP data format.

CTP Shutdown. Shutdown is also shown in Figure 20. 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.

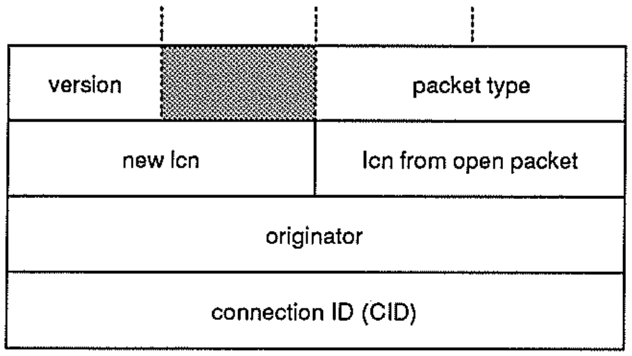


Figure 16: ACK open packet

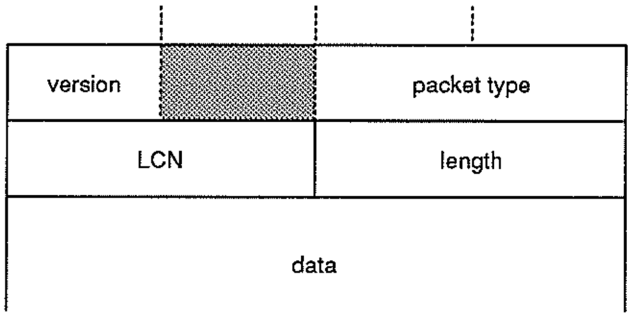


Figure 17: Data packet

4.1.3. 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 4.1.1.

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. COIP-K Module Interchange

4.2.1. Module Introduction. COIP-K has been created to easily design, implement, and test COIP protocols, as was shown in Figure 1. COIP-K is based on the idea that different COIP protocols

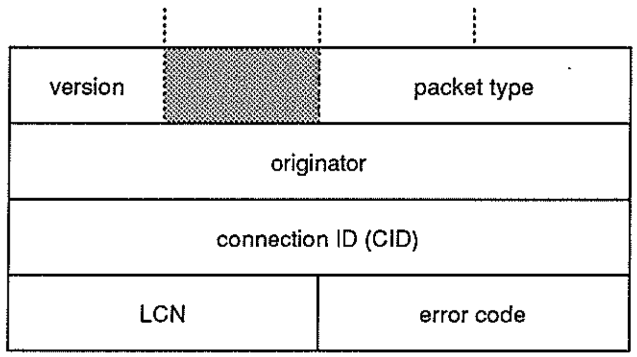


Figure 18: Close packet

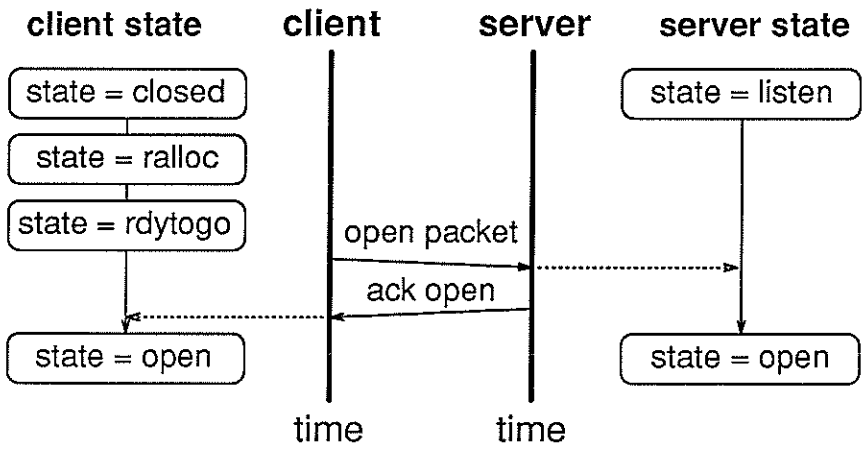


Figure 19: Connection establishment

share 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

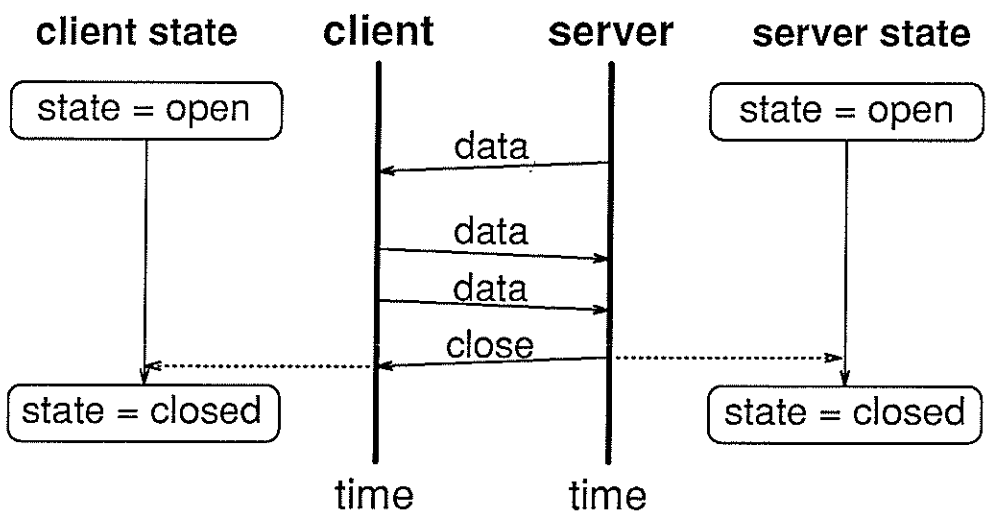


Figure 20: Data transfer and connection closing

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 21 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 22. 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: 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

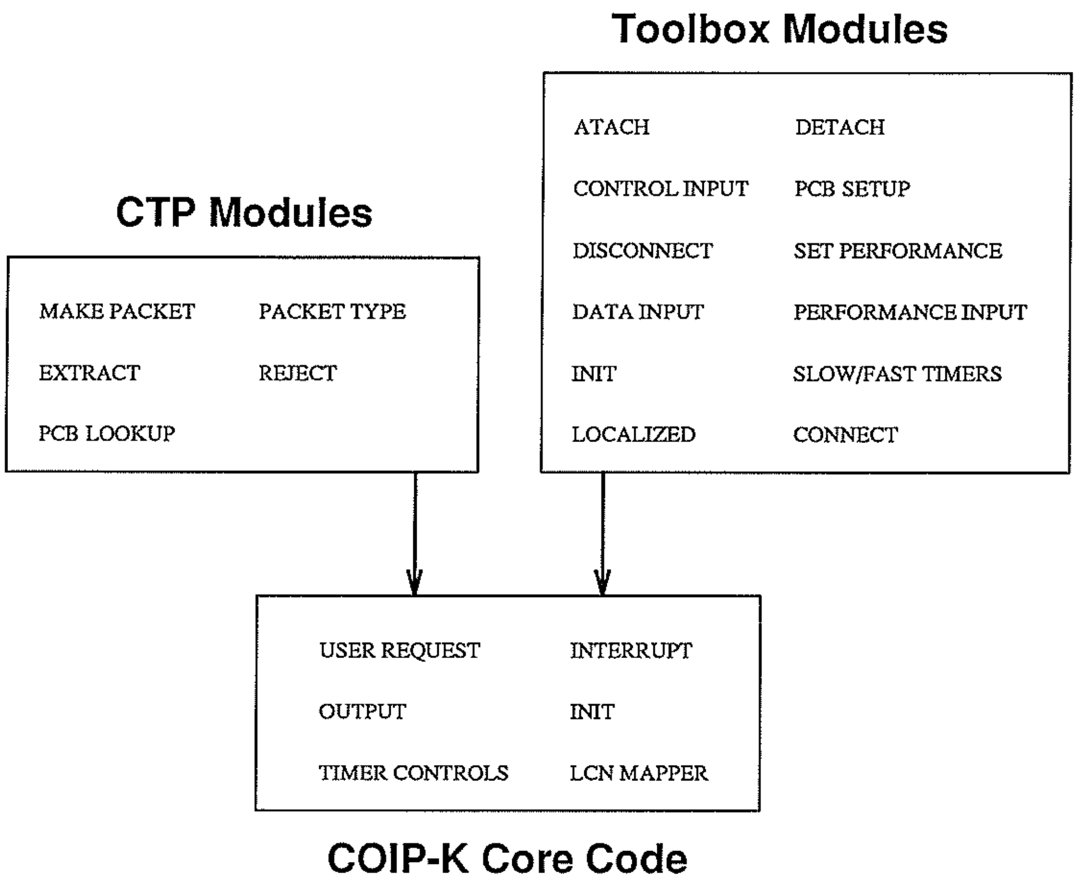


Figure 21: COIP-K plug-in modules

is achieved by COIP-K by dividing the tasks needed for implementing a COIP protocol up into smaller easy-to-write modules and providing example modules to guide a programmer.

4.2.2. 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 23.

To demonstrate the COIP-K module concept, the many-to-many CTP COIP-K module set was

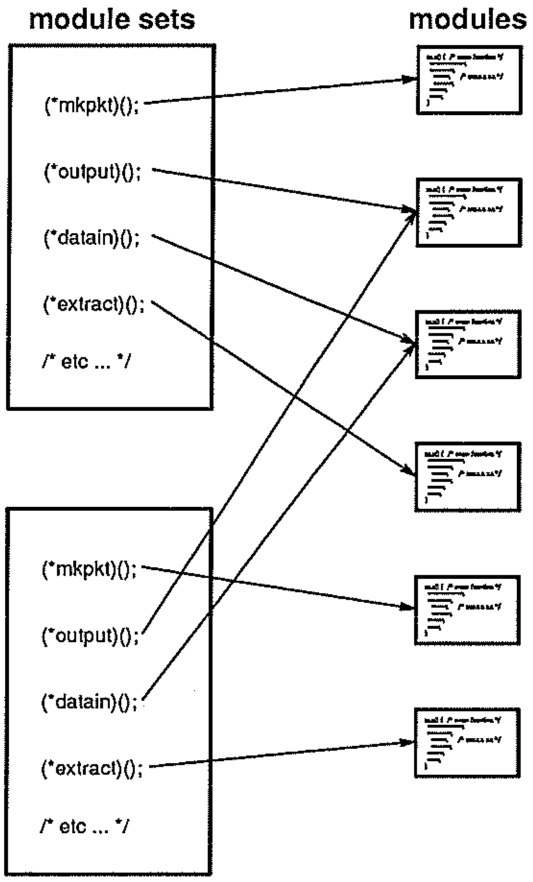


Figure 22: Plugging modules into COIP-K

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 23. 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.

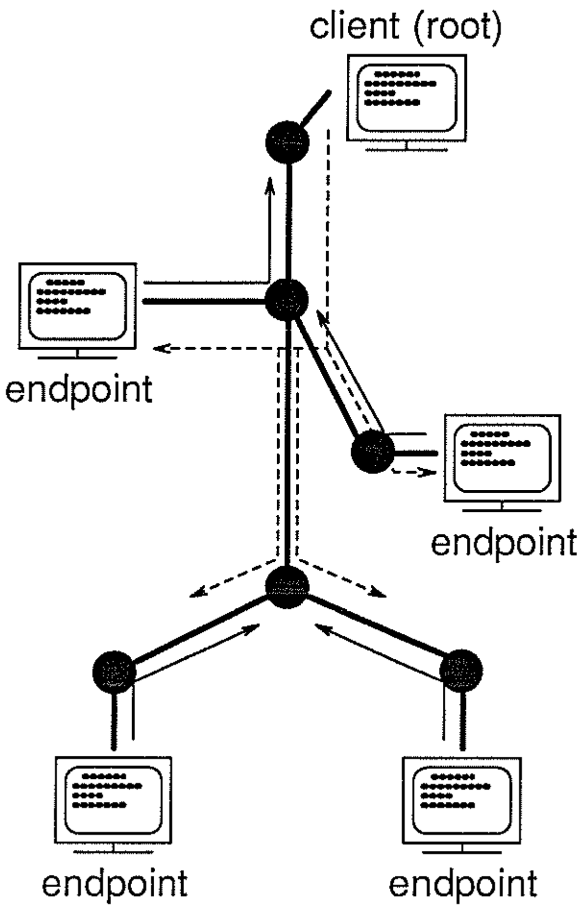


Figure 23: One-to-many multipoint connection

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, a simple multipoint demonstration program 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.

4.2.3. 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.

4.3. 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.

4.3.1. 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 24.

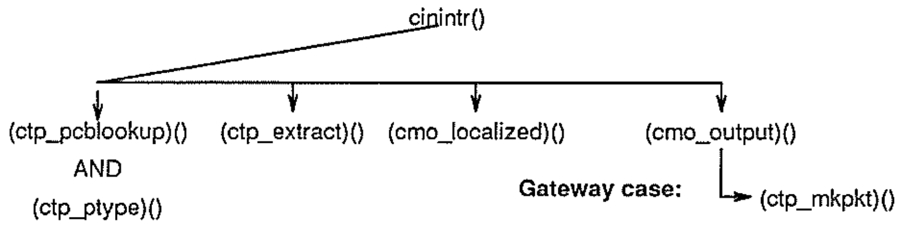


Figure 24: COIP-K critical path

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 certain indirection as shown in Table 1. The table shows the cost of a function call in 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

Table 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

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.

4.3.2. 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[‡]. 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[§]. 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 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 25. 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

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

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

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[¶].

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 25.

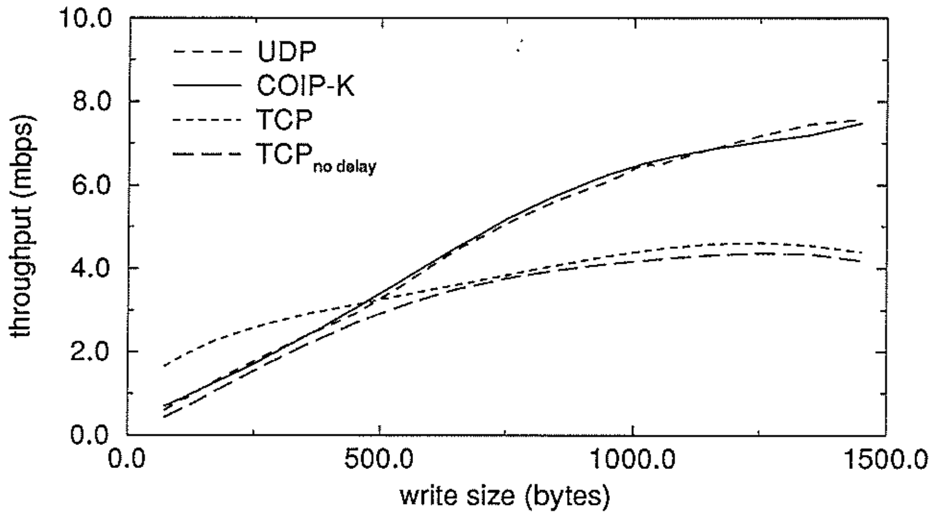


Figure 25: Throughput with TCP_NODELAY option added in

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, we conclude that COIP-K has been implemented efficiently.

4.3.3. COIP-K Delay Performance. To examine COIP-K's delay characteristics, probes were inserted into COIP-K, UDP, and TCP code [13]. 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 26 and 27 respectively. The plots were generated by measuring the processing time of 200 packets, each of which contained 1024 bytes of user-data. Table 2 shows the minimum, maximum, and average delay for the different protocol plots.

[¶]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.

Table 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 μsecs receiving data and 115 μsecs 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.

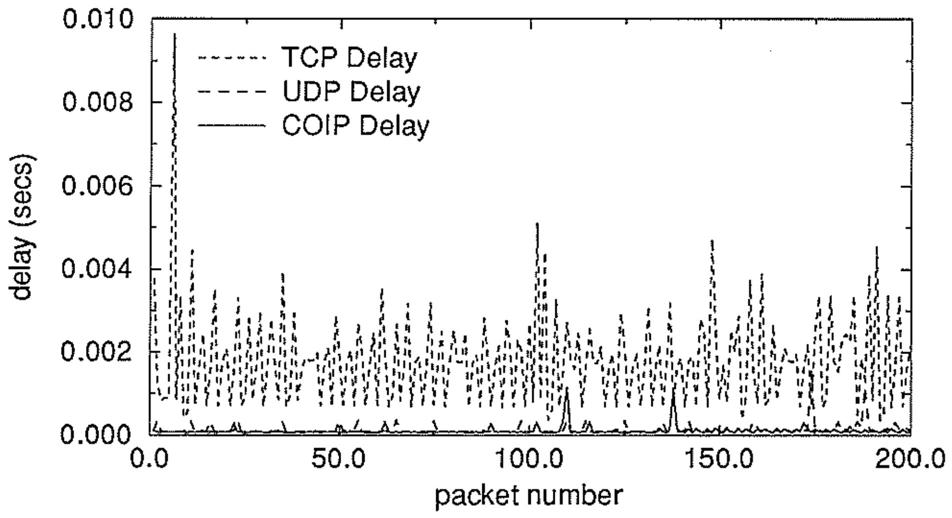


Figure 26: Client delay plot

4.3.4. 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 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.

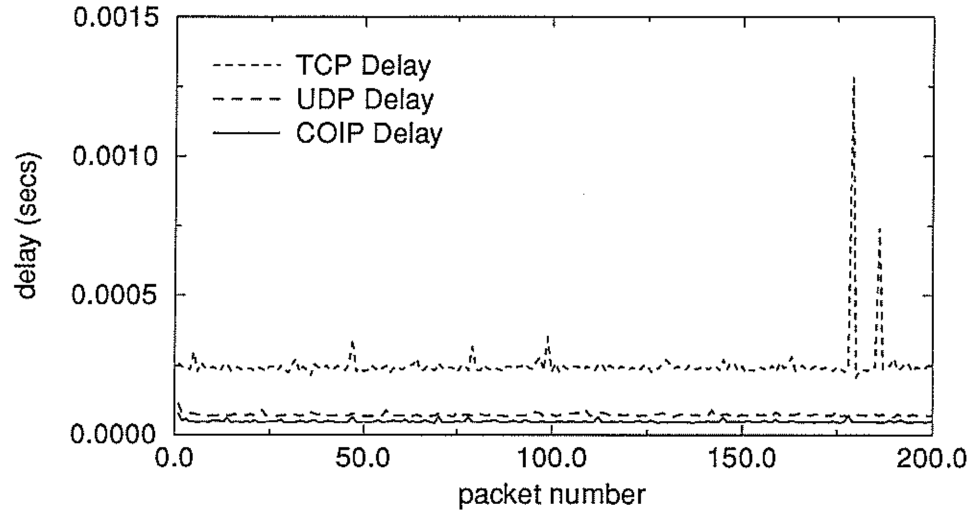


Figure 27: Server delay plot

4.4. COIP-K Testing and Demonstrations

The five 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 28 shows the three main testing configurations: loopback, over a local network, and through a gateway. All applications have been tested in these configurations.

4.4.1. File Transfer Test. The first set of COIP-K test programs COIP-K implement a go-back-n sliding window protocol in user code on top of CTP. The test programs use 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 29. This test generates a lot of COIP events in a short period of time.

4.4.2. 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 30.

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`

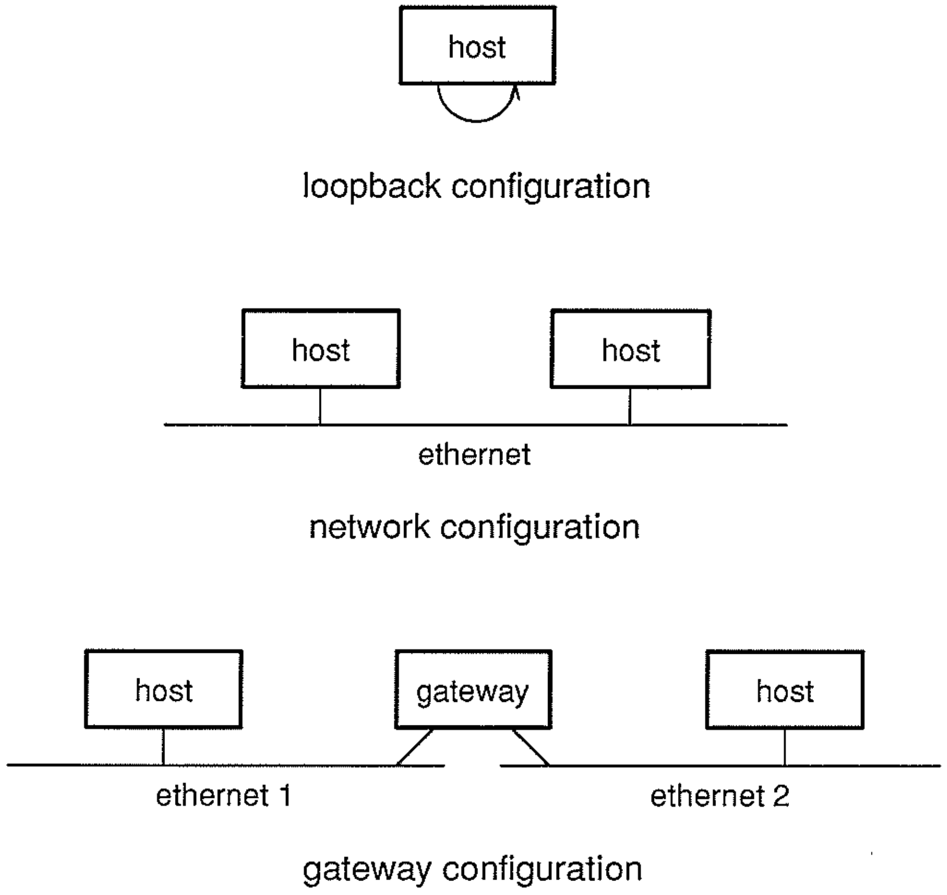


Figure 28: COIP-K testing configurations

program. It has to deal with multiple COIP connections and a large number of COIP packets per second.

4.4.3. 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.

4.4.4. 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 31. Anything a user types is displayed in the input 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.

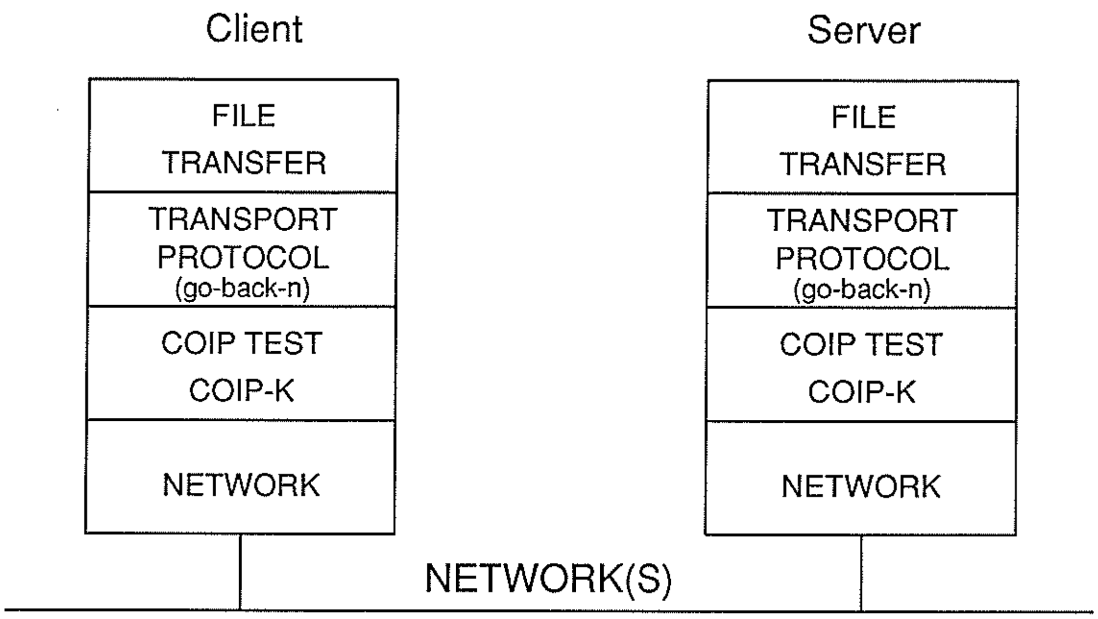


Figure 29: File transfer program

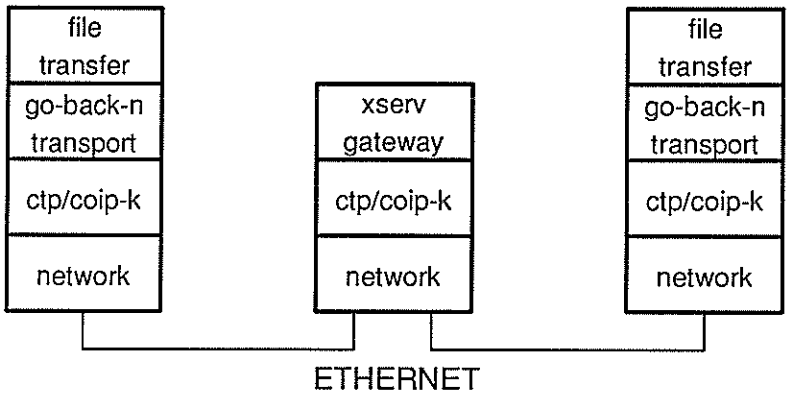


Figure 30: File transfer program, through a third party

4.4.5. 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 32.

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.

5. Conclusions and Future Research

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

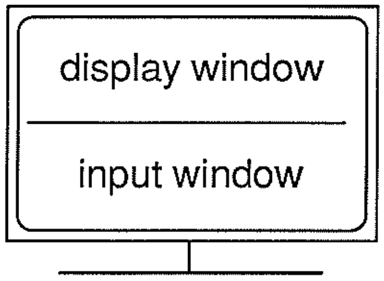


Figure 31: Multipoint chat program screen layout

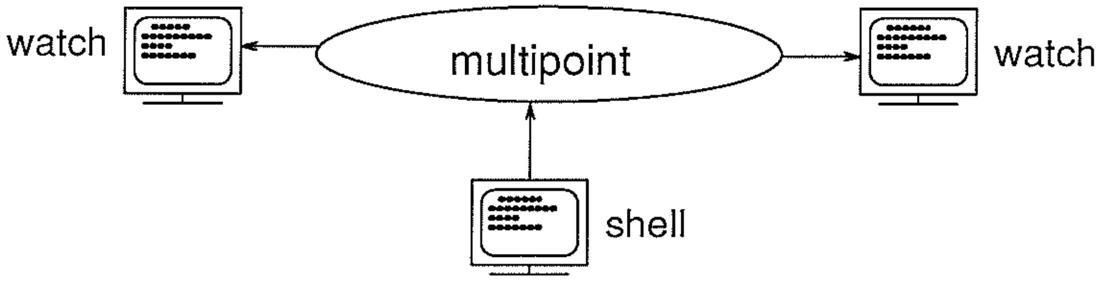


Figure 32: Multipoint script program

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 paper 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.

5.1. Future Research

The research work in this paper 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 ATM interface 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) [16]. 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.

References

[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] Corporation for National Research Initiatives, "Connection IP (cip) Report," *Proceedings of the Twentieth Internet Engineering Task Force*, pp. 109-114, March 1991.

[6] Cranor, C., *An Implementation Model for Connection-Oriented Internet Protocols*, M.S. thesis, Department of Computer Science, Sever Institute of Technology, Washington University, St. Louis, Missouri, May 1992.

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

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

[9] Gross, P., "Connection IP (cip) Report," *Proceedings of the Internet Engineering Task Force*, Ann Arbor, Michigan, October 1988.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Peterson, L., et al., "The x-Kernel: A Platform for Accessing Internet Resources," *IEEE Computer*, vol 23, no 5, pp. 23-33, May 1990.

[16] Sterbenz, James P. and Parulkar, "Axon: Application- Oriented Lightweight Transport Protocol Designm," *Tenth International Conference on Computer Communication*, ICCC, Narosa Publishing House, New Delhi, India, Mov. 1990, pp. 379-387.

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

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

[19] 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.