An Implementation Model for Connection-Oriented Internet Protocols

C. D. Cranor
chuck@maria.wustl.edu
(314) 935-4203

G. M. Parulkar
guru@flora.wustl.edu
(314) 935-4621

Computer and Communications Research Center
Department of Computer Science
Washington University
St. Louis, MO 63130

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 performance trade-offs. A COIP-Kernel which can be used as a toolkit to implement any of the proposed COIPs has been designed. COIP-K features module interchange and incremental software support. The paper presents the COIP-K implementation and its performance characteristics.

1 Introduction

Recently, several research groups have proposed connection-oriented access protocols for the network and internet layers. 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, a connection's resource reservation is enforced throughout the life of that connection. Finally, once a connection's data transfer is completed, the connection is broken and the allocated resources are freed.

These connection-oriented access protocols are perceived to have benefits for applications requiring a variable grade of service with performance guarantees. This has prompted several research groups, including our group at Washington University, to propose connection-oriented internet protocols (COIPs) such as MCHIP [6], ST [3, 8], and FLOW [9].

1.1 COIP-K Motivation

The proposed COIPs have several similarities and differences. The members of the COIP working group of the Internet Engineering Task Force decided that it is important to pursue several protocols and compare and contrast the alternate approaches for implementing them. However, independent implementation of these protocols was considered unwise for the following reasons. First, as the proposed COIP protocols have many common functions, independent implementations would lead to much duplicate work. For example, all the protocols have a connection state machine and a resource allocation and enforcement function. Second, 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. The basic concept of COIP-K is shown in Figure 1.

[1]This work is supported in part by Bellcore, BNR, DEC, Italtel SIT, NEC, NTT, Synoptics, and NSF

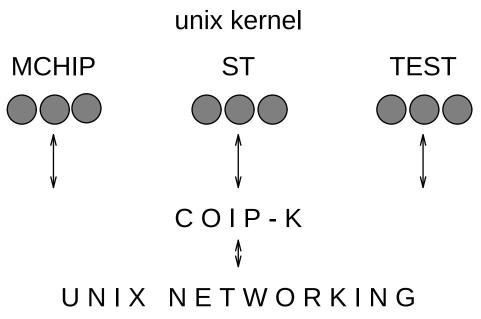

Figure 1: COIP-K structure

This approach to protocol development yields important benefits to research efforts. Many of the functions that a COIP must provide can be supported by alternative mechanisms. These mechanisms can be implemented in experimental modules and integrated with COIP-K to produce different instantiations of COIPs. These instantiations represent mechanisms that can be compared under controlled experimental conditions. As a result, it will be possible to describe the conditions under which each of the alternate mechanisms behaves best, 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 a variable level of support to a protocol programmer. For example, a novice COIP-K user can use mostly default COIP-K functional modules to get a simple test protocol running quickly. As the novice COIP-K user gets more advanced, he or she can swap out more and more default modules in favor of his or her own modules to make a more sophisticated protocol.

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 4.3 BSD interface to 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 can talk to the resource manager for resource availability and allocation, but the actual resource allocation algorithm is part of the resource allocation module. By allowing the protocol-specific functions of a COIP protocol to be in modules, COIP-K can be used to easily implement a wide range of COIPs. It should be noted that COIP-K is a tool, not a protocol. Coip-k is not intended to replace TCP/IP or become a permanent part of the kernel. Instead, it is a tool which can be used to help ease the implementation and exploration of COIP protocols.

This paper describes the COIP-K implementation and demonstrates COIP-K's feasibility and viability.

1.2 Coip-k Implementation Requirements

The scope of this research includes the following major implementation requirements.

- Coip-k must be implemented in the Unix kernel using the standard 4.3 BSD interface.
- Coip-k should allow implementation of various COIP protocols by module interchange.
- Coip-k should refrain, as much as possible, from modifying the user-level socket interface.
- Coip-k should have efficient per-packet processing for high performance.
- Since most COIP protocols support multipoint connections, COIP-K must support them too.

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.

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 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. Coip-k resides in the protocol layer. The third and lowest layer 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. The details of the inner workings of the networking system can be found in other publications [2, 5].

3 COIP-K Implementation

This section describes the implementation of COIP-K within the framework of the Unix networking model.

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. The only change to the standard socket layer interface necessary in our implementation of COIP-K was 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.

3.1.1 Data Structures

The applications programmer of a COIP-K based protocol needs to be familiar with a few new 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 not discussed here because it is considered protocol specific. 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 a zero field (used internally as an offset field). 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 them.

```
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 2: COIP-K multipoint 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 `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 2 shows an example of this. Note that the multiple `cinmad` structures are treated as an array to simplify the programming involved in setting up multipoint connections[2].

3.1.2 Client-Server System Calls

The system call sequence used by COIP-K clients and servers is shown in Figure 3. The call sequences are similar to those of TCP/IP clients and servers. However, before a 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 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()`.

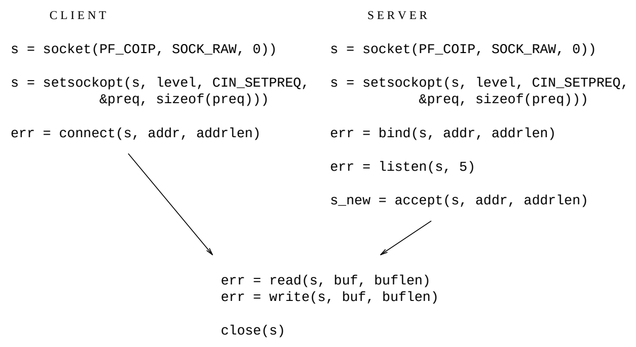
```
CLIENT                                        SERVER

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 3: Sample COIP-K point-to-point client and server

3.2 COIP-K in the Protocol Layer

COIP-K has been designed to work within the BSD Unix networking model. COIP-K resides in the protocol layer of the SUNOS/BSD kernel and has its own com-

[2] Unfortunately some system calls limit the size of an address to the size of an mbuf. Hopefully such restrictions will be removed in the next release of the operating system.

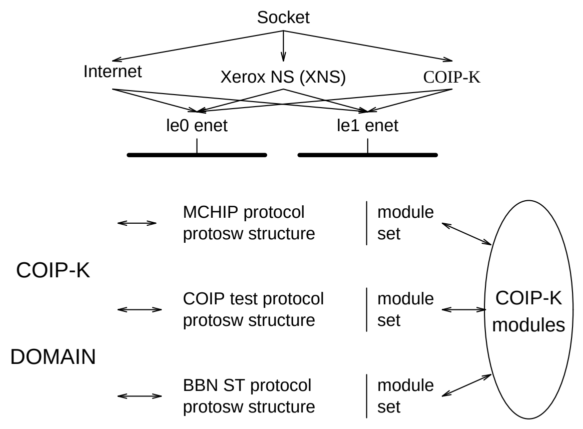

Figure 4: The BSD Unix networking model and COIP-K

munications domain, as is shown in Figure 4. COIP-K has its own 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 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. 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 and Figure 4. The first part is the core COIP-K code which is common to all COIP-K protocols. The second part is a set of modules which are plugged in using a module set on top of the core code to form an implementation of a COIP protocol. A module set may include protocol-specific modules as well as default modules. A COIP protocol built with COIP-K can support connection-oriented[3] communications with resource allocation, packet forwarding/gatewaying, and multipoint connections. The COIP-K code assumes that IP addresses will be used. Also, by default, COIP-K uses IP routing. This can be overridden if the need arises.

[3]Note that the "connection" is not a reliable connection and can provide both connection-oriented and datagram access, thus it is sometimes called a "congram" [7].

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). The COIP-K PCBs are stored in a circular linked list which can be traversed by starting with the address of `cin_q` (a dummy PCB) and following the `p_next` pointer. 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 protocol-specific module will be called every time a COIP-K PCB is created or freed.

The COIP-K state variable indicates the state of the corresponding connection (e.g. CLOSED, OPEN, OPENING, etc.) to COIP-K. 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 long 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.

3.2.2 Multipoint Addressing and Routing in the PCB

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 5. This system was chosen because it interfaces easily with 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.

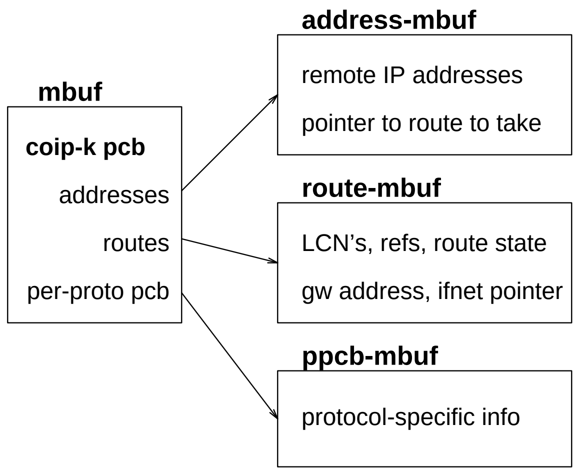

Figure 5: Multipoint PCB structure

3.2.3 Major Functions

This section presents an overview of the main functions of COIP-K. Figure 6 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 6. 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 performs many 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 the 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.

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. This function is the most important function for per-packet processing because it controls packet forwarding.

The interrupt function is scheduled to be called by the network interface layer. It is a loop in which packets are 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 used to look up the PCB. Finally, the packet is processed as a data, open, resource, or control packet. Note that the forwarding of a data packet requires minimal processing.

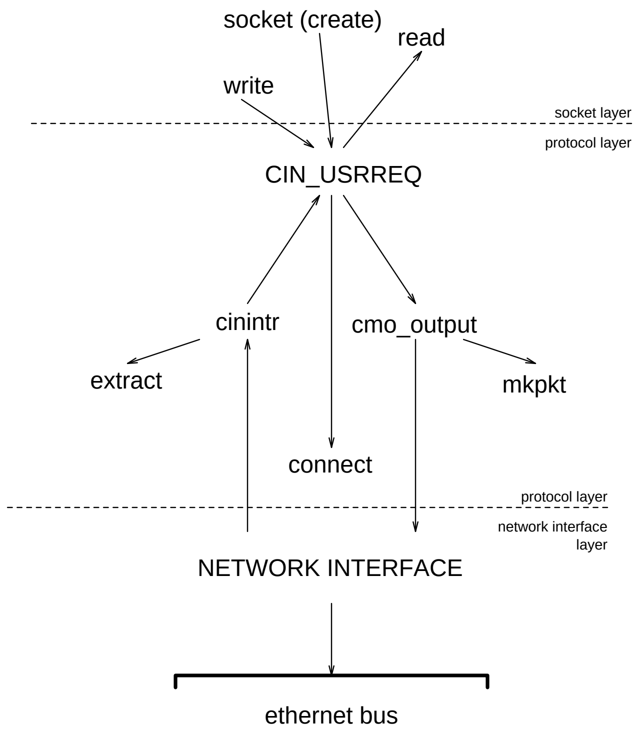

Figure 6: COIP-K protocol switch

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 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.

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. To provide more elaborate connection management, more complex protocol-specific functional modules must be provided.

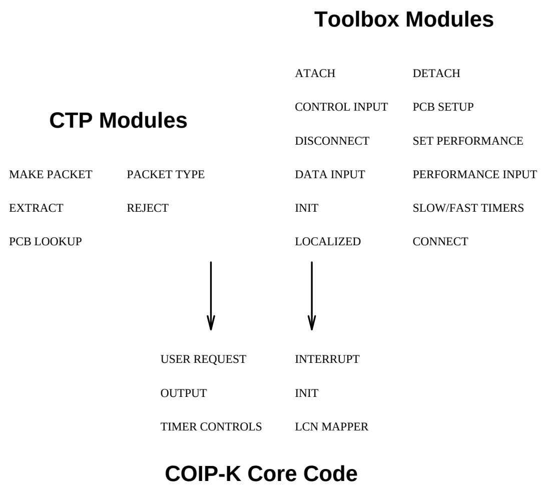

Figure 7: COIP-K plug-in modules

3.3 COIP-K Modules

There are two types of COIP-K modules: required and optional. Required modules are ones that are 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. Figure 7 shows how protocol modules plug into COIP-K for a COIP test protocol (CTP).

By providing COIP-K toolbox modules and allowing them to be interchanged easily with other modules, COIP-K provides a useful feature: incremental protocol development support. When protocol developers start using COIP-K they can use many toolbox modules and have COIP-K do most of the work. As the developers become more experienced with COIP-K and kernel programming they can swap out toolbox modules and replace them with modules of their own. Eventually they may swap out most of COIP-K and replace it with their own code. COIP-K provides as much (or as little) support as the protocol developer needs. The required and optional modules that make up a COIP-K protocol are called 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 removes vital bits of information from 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 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)

COIP-K has many optional modules [2]. They include modules to create and delete protocol-specific information in the pcb, timer modules, module hooks for specifying performance requirements and handling resource, modules for special handling of data and/or control packets, a packet output module, a pcb setup module, and a module called at system bootup time to set up protocol specific data structures.

3.4 COIP-K Module Interchange

The implementation of a COIP protocol with COIP-K consists of two parts: the protocol-specific modules and the common COIP-K code and default modules. The most important feature of COIP-K is the efficient and easy module (protocol function) interchange. This interchange allows multiple COIPs to reside in the kernel and share protocol modules. At the same time, the module interchange feature also allows each protocol to use its own specific modules if necessary. This leads to variable level of support for COIPs from COIP-K as well as easy evaluation of trade-offs associated with different COIPs.

The module interchange functionality is achieved as follows. All modules are implemented as functions and form a common pool of modules. Each COIP protocol includes a module set composed of modules from the module pool. The module set determines the behavior of the COIP protocol. The module set is implemented as a structure with a list of pointers to appropriate modules in the module pool as shown in Figure 8. Important observations on the module selection and building process include:

- Toolbox modules, required modules, and protocol-specific optional modules are all part of the common pool of modules.

- Each COIP has a module set consisting of a list of pointers to a subset of modules in the module pool. A module can be shared among any number of COIPs (*i.e.,* module sets). Module sharing saves space and keeps the kernel size relatively small.

- To create a new instantiation of a COIP, all that has to be done is to create a COIP-K module set by making a copy of the module set structure and setting the pointers appropriately to select the desired subset of modules from the module pool.

- To override a default module, a COIP implementer may add a new module to the module pool and change the appropriate pointer so that it points to the new module.

- A programmer using COIP-K can choose which COIP-K module set 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 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.

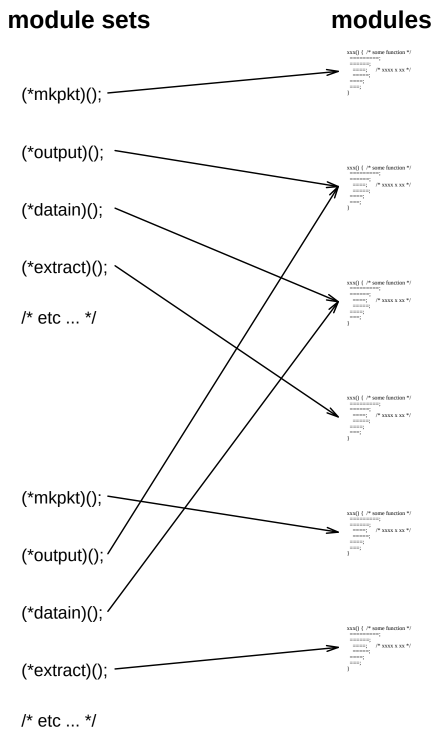

Figure 8: Plugging modules into COIP-K

By providing such an easy-to-use interface at the system call level, COIP-K has made the use of COIP protocols relatively easy. The use of COIP-K should minimize the number of kernel rebuilds and reboots required for COIP testing. In fact, if done carefully, COIP-K modules can be swapped on a running system by changing pointers in kernel memory without rebooting the system.

4 COIP-K Feasibility and Viability

This section presents the feasibility and viability of COIP-K, and how it has been designed to meet the four main objectives outlined in Section 1.2.

4.1 COIP-K Test Protocol

An example COIP protocol, the COIP-K Test Protocol (CTP), has been specified and successfully implemented using COIP-K. This exercise served two purposes. First it has helped demonstrate COIP-K's usefulness in creating implementations of a COIP protocol. Second, it has helped thoroughly debug and test COIP-K. It is important to note that the CTP represents a subset of Washington University's MCHIP protocol. 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, they were not implemented in CTP. Resource allocation and enforcement are protocol specific, and thus should be part of COIP modules and not part of COIP-K itself. The CTP protocol has been intentionally kept simple because the emphasis of this research is on COIP-K. CTP is expected to serve as a template for implementation of other COIP protocols using COIP-K.

Details of the CTP protocol are presented in another paper[2]. The CTP specification includes connection set up, data transfer, connection termination procedures, and suitable packet formats.

4.2 Module Set Demonstration

In order to demonstrate the power of COIP-K in facilitating module interchange, a new COIP protocol, CTP2, has been created. CTP2 and CTP differ from each other in the way each supports multipoint communication.

There are two fundamental ways to do multipoint connections: many-to-many and one-to-many. CTP, like the MCHIP protocol, 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 9. The ST protocol supports the one-to-many paradigm of multipoint communication. CTP2 is the same as CTP for point-to-point connections, but it is one-to-many for multipoint connection.

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 9. The root endpoint should never forward a received packet,

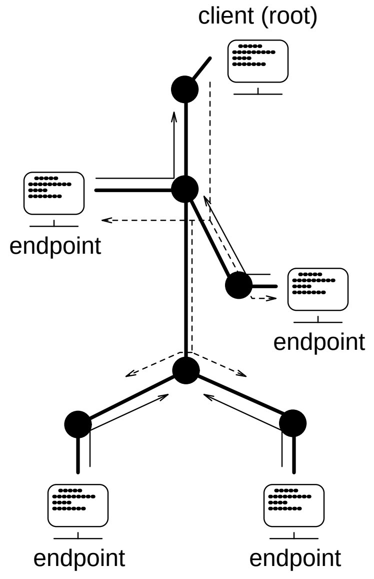

Figure 9: One-to-many multipoint connection

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. CTP2 was developed by first copying the CTP module set.

To handle the packet forwarding requirements of CTP2, the output module had to be modified. The CTP module set used the COIP-K toolbox module `cmo_output` as its output module. For the CTP2 module set a copy was made of the `cmo_output` module. This copy was then renamed to `ct2_output` and modified 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 set to point to the `ct2_output` function.

When the CTP protocol receives a data packet, it passes the data to the socket layer as per the COIP-K default. 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 quickly create CTP2. A simple multipoint demonstration program was recompiled to use CTP2. CTP2 was tested and it worked as expected.

To implement CTP2 from scratch would take a long 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, designing and implementing CTP2 to operate in parallel with CTP took about an hour.

4.3 COIP-K Performance

A study of CTP's performance has been undertaken to verify that COIP-K works properly and to quantify the performance of COIP-K. In terms of delay and throughput, COIP-K is similar to UDP and better than TCP. COIP-K also was found to have a minimal cost due to its use of indirection in function call. More details on the performance can be found elsewhere [2].

4.4 COIP-K Testing and Demonstrations

The COIP-K test applications include: a go-back-n file transfer program, a ported version of `telnet`, a multipoint chat program, and a multipoint script program (for session logging). COIP-K 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 multipoint applications. The same test programs can exercise different parts of the COIP-K code depending on what hosts are members of the connection. The three main testing configurations are: loopback, over a local network, and through a gateway. All applications have been tested in these configurations.

5 Conclusions

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.

COIP-K was tested and shown to be both feasible and viable. The five main implementation requirements set for COIP-K were met.

References

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

[2] 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.

[3] Forgie, J., "ST - A Proposed Internet Stream Protocol," IEN119, MIT Lincoln Laboratory, 7 September 1979.

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

[5] 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.

[6] 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.

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

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

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