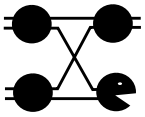

Node Controller Managed Object (NCMO) and Node Controller Communication Protocol (NCCP)[1]

Dakang Wu
John D. DeHart

Version 1.0
Applied Research Laboratory
Department of Computer Science
Washington University
St. Louis, Missouri 63130
dw1@arl.wustl.edu

Working Note ARL-96-03
July 3, 1996

Abstract

With the development of ATM technology and increasing deployment of ATM networks, we anticipate a heterogenous environment for an ATM network. Switches and client stations from different vendors, each with potentially different control mechanisms, will be used within the same network. This diversity of control structures introduces great complexity into the development of ATM control software. In this and associated other documents, we propose a software architecture that manages this heterogenous environment. A key aspect of the software design is that the hardware details of a switch and its control mechanism is encapsulated in a low level software module called the Switch Controller (SC). The Node Controller Communication Protocol (NCCP) is presented that allows higher layer software modules to communicate with the SC. The NCCP is general enough to support general multipoint-to-multipoint communications. A general interface to the NCCP, the Node Controller Managed Object (NCMO) is also presented. The NCMO is an Application Programming Interface (API) to the NCCP for the higher level software modules. The development of the NCMO and the NCCP allows the higher layer modules to operate on an abstract switch model and not have to understand the details of every possible hardware switch that might be present in the network. This partitioning of functionality provides a clean interface between software modules and hence, a viable software architecture for the control of a heterogenous set of switches.

1.This document was originally part of the GBNSC document, the first draft of which was written by Ken Cox.

1. Introduction

In this document, we present the design and some implementation details for the Node Control Communications Protocol (NCCP) and the Node Control Managed Object (NCMO). These two object oriented software libraries are integral parts of our implementation of the Connection Management Software System (CMSS) [7]. They provide the objects and communications for CMSS.

Section 2 briefly introduces the design of the CMSS. Section 3 describes the abstract switch model that the NCCP and NCMO support. The NCCP is presented in Section 4 and the NCMO is described in Section 5.

It is highly recommended that the Connection Management Software System (CMSS) Architecture document [7] be read prior to continuing with this document.

2. The Core CMSS

We anticipate that the heterogeneity of ATM switching network equipment will contribute greatly to the complexity of controlling these networks. As shown in Figure 1, network switching systems will be produced by various vendors and (although all will presumably conform to the appropriate ATM standards) may differ considerably in their control requirements and protocols. The same will undoubtedly be true of the client terminals connected to the network. A third source of network heterogeneity lies in the links connecting the switches and the terminals, which will vary in bandwidth and the capability to support QOS requirements.

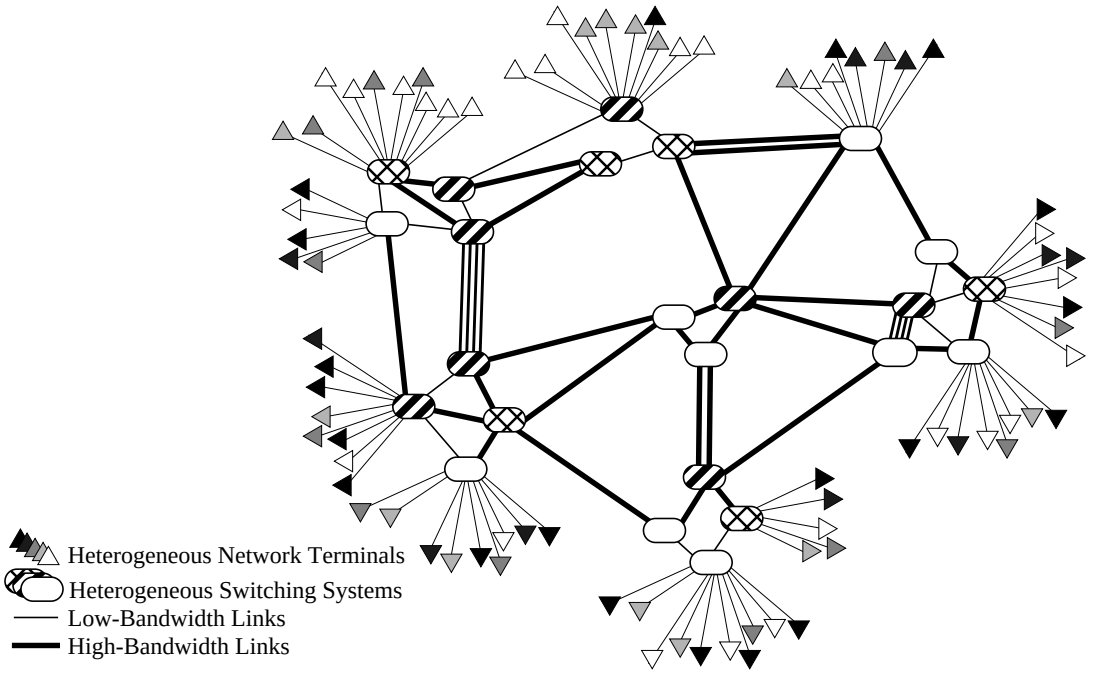

Figure 1. Heterogeneous ATM Networks.

Managing such networks requires to introduce a number of control abstractions which serve to encapsulate and conceal the differences in equipment. Figure 2 illustrates a number of these abstractions. A *node* abstracts a set of one or more interconnected switches, providing a view of the group as a single large switch with a known interface and capabilities. A *control processor* (CP) is an abstraction of the control software for a single node; in some cases the software may actually run on a single machine, while in others it may be distributed. In our control model, this software is the Core Connection Management Software System (Core CMSS) [7]. The control software for the nodes must communicate to set up inter-nodal connections. This communication is in accord with specified, uniform *inter-*

nal protocols (*e.g.*, CMNP[8] in our system). Finally, the terminal-network control interface is encapsulated by *access protocols* which specify the manner in which clients request connections through the network. Examples of such access protocols would be CMAP [6] and Q.93b [1,2,4].

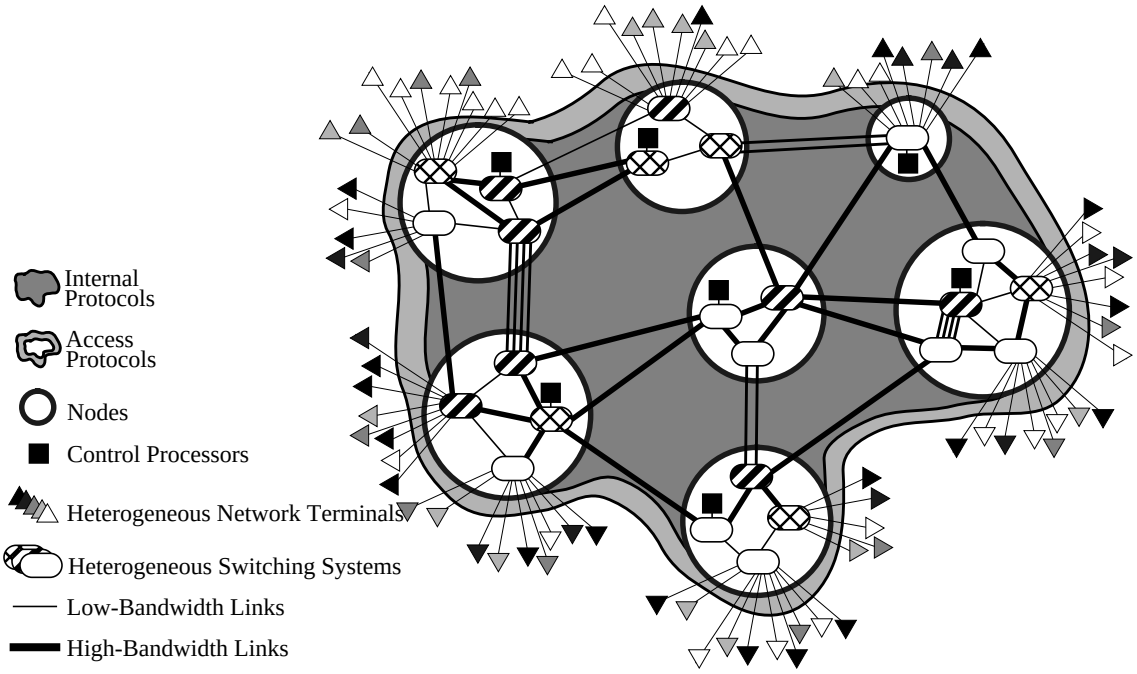

Figure 2. Control Abstractions for ATM Networks.

The Core CMSS present at each node is structured in three layers. The *Connection Management Layer* is actually distributed across all the nodes of the network, and uses the internal protocols of Figure 2 to set up inter-nodal connections. At each node, the Connection Management Layer communicates with the *Node Management Layer* for that node. The Node Management Layer abstracts the collection of switches in the node so that they "act like" a single large switch to the Connection Management Layer, and is responsible for managing intra-nodal connections within the node. It issues commands to the *Switch Management Layer*, which handles connections within the individual switches and conceals hardware dependencies from the other layers.

In the current design, the Core CMSS is realized as a tree of processes (or threads or tasks) as shown in Figure 3. The top process in the tree is the *Connection Manager* (CM) for the node. This process communicates with the CMs of other nodes and with one subsidiary process. For the node shown in the figure, where three switches are grouped and managed as a single node, the subsidiary process is a *Node Controller* (NC). This NC in turn communicates with its subsidiaries; in the example these are one *Switch Controller* (SC) for each switch. If, as in the example, the switch hardware is supplied by several different vendors, these SC processes will be from different executable, each tailored to control the specific hardware. However, the API provided by each SC is identical, and is identical to the NC API. The GigaBit Network Switch Controller (GBNSC) [10] is one example of an SC.

CMSS processes communicate only along the links of the process tree. For example, in Figure 3 the CM and NC may communicate and the NC may communicate with each SC. However, the CM does not communicate directly with the SCs, nor do the SCs communicate with one another — indeed, the encapsulation provided by the CMSS is such that these processes have no knowledge that the others exist.

This design decision — that the SC and NC have the same API — was originally motivated by the desire to have a node "look like" a switch to the layers above the node. However, it also has some nice consequences for the setup and management of the process tree. As shown in the examples of Figure 4, the upper example shows a single switch

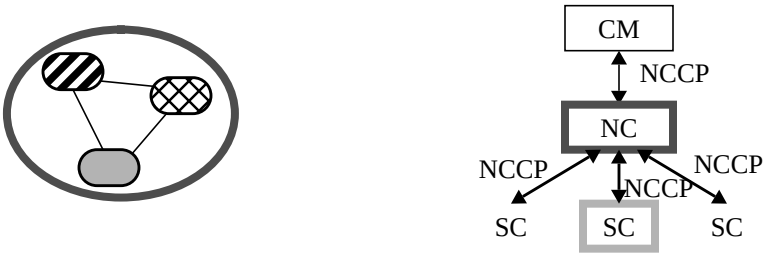

Figure 3. Example CMSS Process Structure.

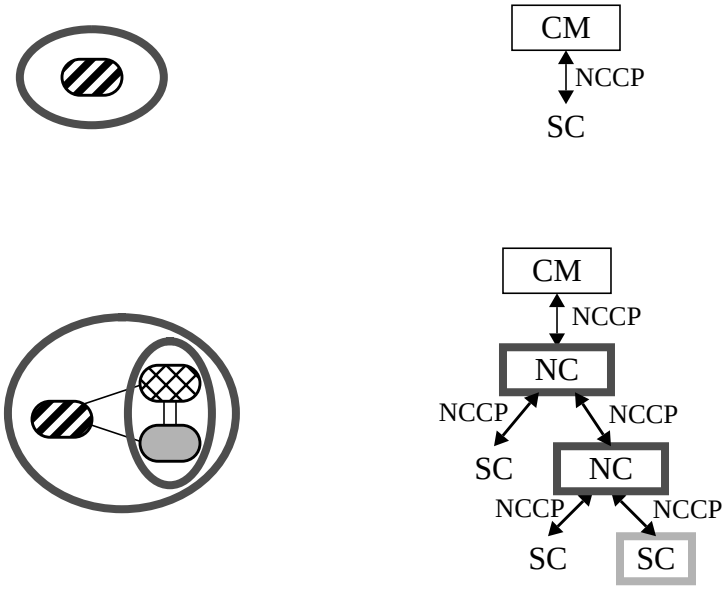

Figure 4. Additional CMSS Process Structures.

which is to be managed as a node. In this case we can omit the Node Management Layer entirely, and have the CM communicate directly with the SC for the switch. The lower example shows a node hierarchy, where one of the subsidiary elements of a node is another node. The process tree reflects the nested structure, and because the NC and SC APIs are identical the upper NC is unaware that one of its subsidiaries is actually a multiple-switch node.

The API used by the NCs and SCs is implemented as a software object called the *Node Controller Managed Object* (NCMO). The NCMO encapsulates the interactions between two processes (a parent and a subsidiary) in the CMSS process tree. When a parent process determines it has a subsidiary, it creates an NCMO object for that subsidiary. The creation of this object causes the creation of the subsidiary process and the communications links to that process. Subsequent interactions with the subsidiary are handled through function calls on the NCMO. These calls typically cause communication with the subsidiary; the NCMO uses the *Node Controller Communications Protocol* (NCCP) (Section 4) for this communication.

The NCMO provides the user a view of an abstract switch with abstract objects and a set of abstract operations. The user of NCMO can create abstract objects and perform abstract operations through NCMO. The NCMO, besides organizing its own objects, will sent the operation request to the subsidiary through the NCCP. The subsidiaries will implement these abstract operations in real physical environment.

3. Abstract Switch Model

Figure 5 shows an abstract switch model presented by the NCMO. This abstract switch is modeled as a black box with a number of links. An abstract switch has a switch Id given by its CMSS parent. Each link of the switch is identified by a unique integer identifier. A link may have different capacity in each of its two different directions.

A link has a remote end identifier which indicates what entity is connected to that link. The remote end may be another switch port or it may be a terminal. Terminals are identified in the figure by *T(“name”)*. If the remote side is a switch port, it is identified by a tuple (CM address, switch Id, other side local linkIdentifier). Each port also has several resource parameters, such as the available bandwidth and VPI and VCI ranges. The remote end information and resource information could be obtained by the switch controller through a Hello protocol [12] and/or configuration files.

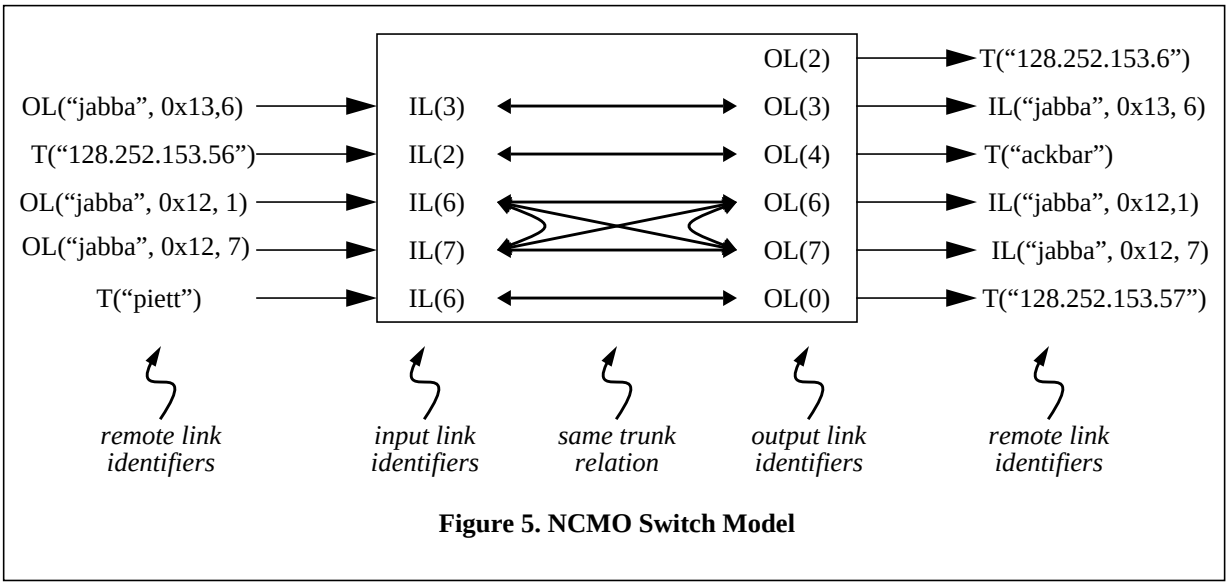

Figure 5. NCMO Switch Model

The switch model provides a same-trunk relationship between the links. Links typically have this relationship if they are connected to the same remote entity, i.e. the same terminal or the same switch. This relationship is used in conjunction with the echo parameters of sinks (described below) when setting up connections.

The abstract switch is treated as a large ATM crossbar in which it is possible to route any input data stream (i.e., any VP or VC on any of the input links) to any output data stream (any VP or VC on any of the output links).

A set of abstract objects are used within an abstract switch to build, maintain and manipulate connections. A *MultiPoint* (MP) is an abstract object that represents a connection. Since a connection can be either Virtual Path (VP) or Virtual Channel (VC), the multipoint object comes in two varieties, *VPMultipoint* (VP MP*)* or *VCMultipoint* (MP). An MP has a set of attributes, such as the bandwidth requirement of the connection; the QOS requirement etc. A MP also maintains a list of sources and sinks. A *Source* is an abstract object that represents an input of the connection. A *Sink* is an abstract object that represents an output of the connection. In the abstract switch model, any data received on a source will be sent to all the sinks in the MP. The most important attribute of a source or a sink is its VXI (VPI and/or VCI based on connection type). In the source, the VXI identifies an input channel. In the sink, the VXI identifies an output channel. Source and sink objects, like multipoint objects, can be either Virtual Path or Virtual Channel. A list of attributes of MP, sources and sinks is given in Table 1.

Given an input link, the CM (or more generally the CMSS parent) may create a VPSource object or a VCSource object associated with the link. The *VPSource* represents a source of ATM cells arriving on the link and using VP routing, while the *VCSource* represents an ATM cell stream using VC routing. Similarly, the CM can create *VPSink* and *VCSink* objects from output links. The CM may also create VPMultipoint and VCMultipoint objects.

Table 1: Properties of NCMO Switch-Model Objects

Object	Parameter	Description	Assignment
VPSource	VPI	VPI of the cell stream	CM or subsidiary
VPSink	VPI	VPI of the cell stream	CM or subsidiary
	echo	Should sink receive cells from sources on the same trunk	CM (default FALSE)
VCSource	VPI	VPI of the cell stream	CM or subsidiary
	VCI	VCI of the cell stream	CM or subsidiary
VCSink	VPI	VPI of the cell stream	CM or subsidiary
	VCI	VCI of the cell stream	CM or subsidiary
	echo	Should sink receive cells from sources on the same trunk	CM (default FALSE)
VPMultipoint	Bandwidth	Connection's bandwidth (peak, average, etc.)	CM (default best-effort)
	Priority	Connection's priority	CM (default low)
VCMultipoint	Bandwidth	Connection's bandwidth (peak, average, etc.)	CM (default best-effort)
	Priority	Connection's priority	CM (default low)

Once the CM has created source, sink, and multipoint objects, it may connect them together. VP objects may only be connected to other VP objects, and VC objects may only be connected to other VC objects. {I think we should add a description of how we would terminate a Virtual Path and break out the VCs.} Sources and sinks can only be connected to multipoints. Figure 6 gives some possible connections that might be constructed for the abstract switch in Figure 5. The VPI, VCI, and echo parameters of the sources and sinks are shown where appropriate. The "semantics" of such interconnections are as follows:

- Cells arriving at a source (i.e., on a particular input link with the VPI or VPI/VCI described by the source) are delivered to all multipoints connected to the source. Each multipoint gets a separate copy of each cell, and the cells are delivered to the multipoint in the same sequence in which they arrived at the source.

- Cells delivered to a multipoint by the sources are delivered to all sinks connected to the multipoint. Each sink gets a separate copy of each cell, and all cells from a particular source are delivered to the sink in the same sequence in which they arrived at the multipoint; however, the interleaving of cells from different sources may differ at different sinks.

- Cells delivered to a sink by the multipoints connected to the sink are sent out on the sink's link with the VPI/VCI associated with the sink. Cells arriving from a particular multipoint are sent out in the same order in which they arrive at the sink; however, the interleaving of cells from different multipoints may differ at different sinks. If the sink's echo parameter is FALSE and the cell originated from a source that is

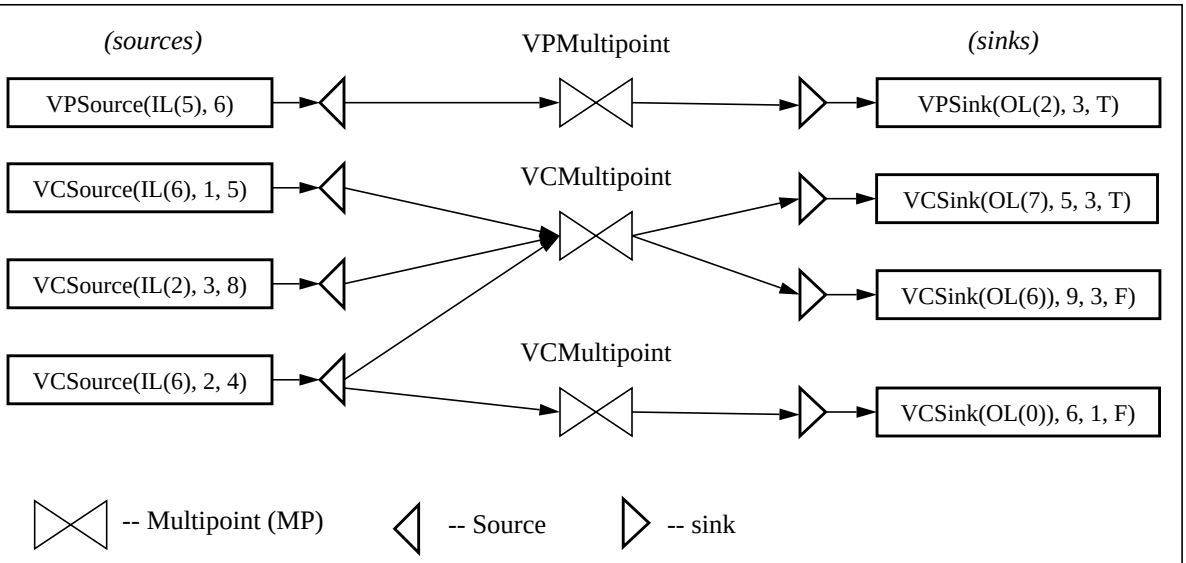

Figure 6. NCMO Connections of Sources, Sinks, and Multipoints

related to the sink by the same-trunk relationship, the cell is discarded and not sent out on the link.

- The ATM standard is followed for VP connections, in that the VCI field of a cell arriving at a VPSource and passing through a VPMultipoint and VPSink is not changed by the switch.
- Cells which arrive at the switch but do not correspond to any source are discarded.

Several examples using the connections in Figure 6 may help to illustrate this.

1. A cell arriving on link IL(5) with VPI=6, VCI=X will be transmitted out of the switch on link OL(2) with VPI=3 and VCI=X, for any X.
2. A cell arriving on link IL(6) with VPI=1, VCI=5 will be transmitted out of the switch on link OL(7) with VPI=5, VCI=3. It will not be transmitted out of link OL(6) with VPI=9, VCI=3 because the source IL and sink OL are related by the same-trunk and the sink echo parameter is FALSE.
3. A cell arriving on link IL(6) with VPI=2, VCI = 4 will be transmitted out link OL(7) on VPI=5, VCI=3; and out OL(0) with VPI=6, VCI=1 via the second VCMultipoint.
4. A cell arriving on link IL(5) with VPI not equal to 6 will be discarded (assuming the connections shown are the only ones that exist). Likewise, a cell arriving on link IL(6) with VPI/VCI not equal to 1/5 will be discarded.

4. NCCP

The Node Controller Communications Protocol (NCCP) is a message-based protocol by which two processes in the Core CMSS tree associated with a node, communicate with one another. Processes outside the CMSS tree may also need to communicate with the NCs or SCs. For example, for network management purposes it might be useful to have a tool that allows the network administrators to issue commands directly to an SC, bypassing the CM and NC. *Jammer* [3], also developed by ARL, is an example of such a tool. The user of Jammer may interact directly with the SC to examine or modify any of the switch tables, to set up connections, and to execute test suites that verify switch functionality. To meet these communication requirements, the NCCP is designed to be composed of a Core NCCP (C-NCCP) and some Expandable NCCP (E-NCCP). The C-NCCP defines the general NCCP message format, base

objects that implement NCCP and the basic messages that support the implementation of operations on the abstract switch. The E-NCCP is the expansion of NCCP that supports the communications for specific SC's. For example, the GBNSC E-NCCP supports the communications between Jammer and the GBNSC, which controls a GigaBit Network switch [9] being built at Washington University. In this document, we only introduce C-NCCP. E-NCCP is specified in other documents[3, 10].

The setup of the communications links is handled at process creation time and is not a concern of the NCCP. The NCCP requires lossless, message-based communication in which ordering of messages is maintained. In the current implementation, PriquePairs [7], which are shared memory based communication objects, are used within the CMSS tree and BTP [7] is used between Jammer and the SC.

Each NCCP operation has a *request phase*, in which one process sends a request message to another, and a *response phase*, in which the second process replies with zero or more response messages. The number of responses depends on the operation being performed. We use the term *alert* to refer to a communication in which no response messages are sent. Some example uses of the NCCP are:

- To let the parent know the status of a link, a SC (or NC) periodically sends LinkStatus message to its parent. The parent can use this information to update its topology and routing databases. The parent does not send a response to this request (the LinkStatus message is an *alert*).
- An NC determines that a subsidiary (which may be an NC or an SC) must create a VP source on a particular link and bind it to a particular Multipoint. It sends a ReserveMP request which contains the MP identifier and the VP source information. The subsidiary returns a single response message which indicates whether the operation succeeded.

NCCP communications are asynchronous. After sending a request, a process can continue with other work while it is waiting for the response(s) to arrive. If a process sends two or more requests, the responses may arrive in any order or be arbitrarily interleaved (however, the responses to any one request will still be ordered as required by the operation).

4.1. Core NCCP

The Core NCCP defines the general format of NCCP messages. A NCCP message is composed of a fixed size NCCP header and a variable size NCCP body as shown in Figure 7.

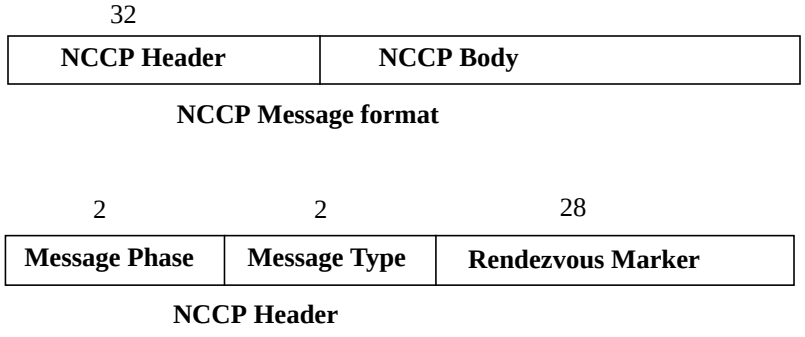

Figure 7. NCCP Message and NCCP Header

The *Message Phase* field indicates this is a NCCP_Request or a NCCP_Response message. The *Message Type*

identifies the message. In the C-NCCP, five types of messages are defined as listed in Table 2. The *Rendezvous*

Table 2: Messages defined in C-NCCP

Message Type	Request Direction	Response Needed	Description
NCCP_ChildStatus (1024)	C-P	No	Report the child is up
NCCP_LinkStatus (1025)	C-P	No	Report link Up/Down status
NCCP_ReserveMP (1026)	P-C	Yes	Request to reserve a Multipoint
NCCP_UpdateHardwareMP (1027)	P-C	Yes	Request to set up hardware tables
NCCP_RollbackMP (1028)	P-C	Yes	Request to go back to previous committed state

Marker is a rendezvous object [7] which provides a safe mechanism to indicate which object is responsible to process the response message if any.

4.1.1. NCCP_ChildStatus Message

A NCCP_ChildStatus message is sent by the child process in the CMSS tree to its parent to report the child process status.

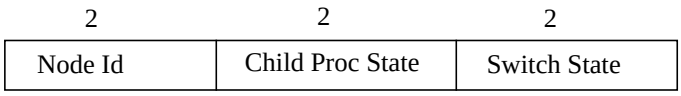

Figure 8. NCCP_ChildStatus Message Format

The *Node Id* is an unsigned integer that indicates the source of the message. The *Node Id* is given by the parent process when it starts the child process. The *Child Proc State* field takes a value from Table 3.

Table 3: Child Process State

Value	Description	Value	Description
Not_Ready (0)	not usable	BadConfig(5)	wrong config file
NoFork (1)	unable to fork child	BadHardware(6)	hardware problem
ChildDied (2)	child process died	BadIO(7)	I/O err
TimeOut (3)	time out	MemoryErr(8)	memory allocation err
NOConfig (4)	no config file	Ready (10)	Ready to work

The *Switch State* field takes one of the two values: SwitchStatus_Dead or SwitchStatus_Alive.

4.1.2. NCCP_LinkStatus Message

A child process periodically sends a NCCP_LinkStatus Message to its parent to report the status of a link. LinkStatus message contains all the link information that the higher level should know.

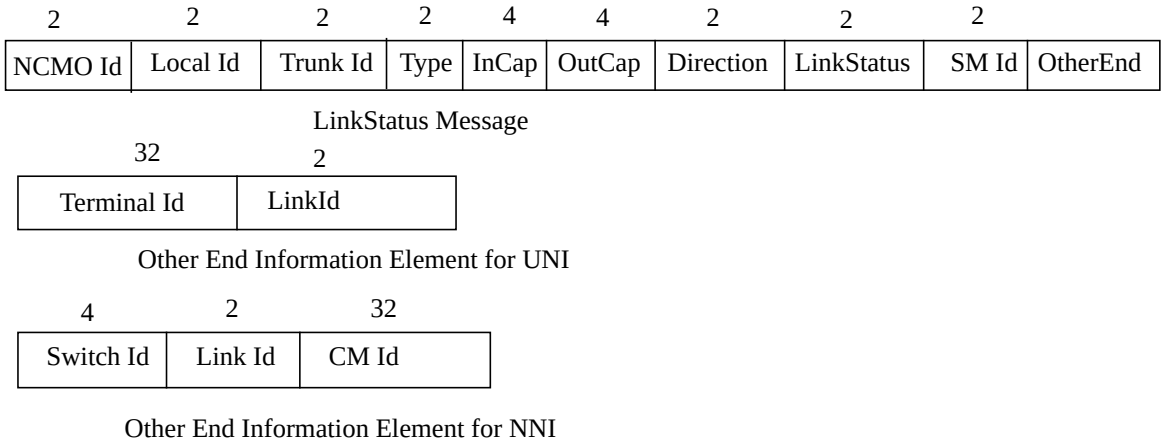

Figure 9. LinkStatus Message Format

NCMO Id is an unsigned integer which identifies the NCMO object. *Local Id* is an integer identifying the link for the abstract switch. *Trunk Id* is reserved for future use. *Type* indicates the link is NNI or UNI. *InCap* and *OutCap* are four byte integers to give the capacity of the link. *Direction* and *LinkStatus* are *SC_LinkDirection* type fields defined in SC_Common.h. *SC_LinkDirection* can take values listed in Table 4. These two fields together give the link direction and link status. *SM Id* identifies which session manager is responsible for the link. The *Other End* information element has two formats. If the link is a UNI link, it has a terminal identifier, which could be a machine name, and an integer link identifier to distinguish multiple links connecting the same terminal. If the link is an NNI link, it contains the link identifier at the other side that includes the switch controller identifier, link identifier and the Connection Manager identifier, which could be the remote CP's machine name.

Table 4: SC_LinkDirection

Value	Description
SC_LinkDirection_NULL (0)	No Link available
SC_LinkDirection_ToSwitch (1)	Input Link
SC_LinkDirection_FromSwitch (2)	Output Link
SC_LinkDirection_Bidirectional (3)	Bidirectional

4.1.3. NCCP_ReserveMP Message

A NCCP_ReserveMP message carries the information to let the child process reserve resources for a multipoint.

It contains the multipoint information and the information of sinks and sources in the multipoint. The message format is shown in Figure 10.

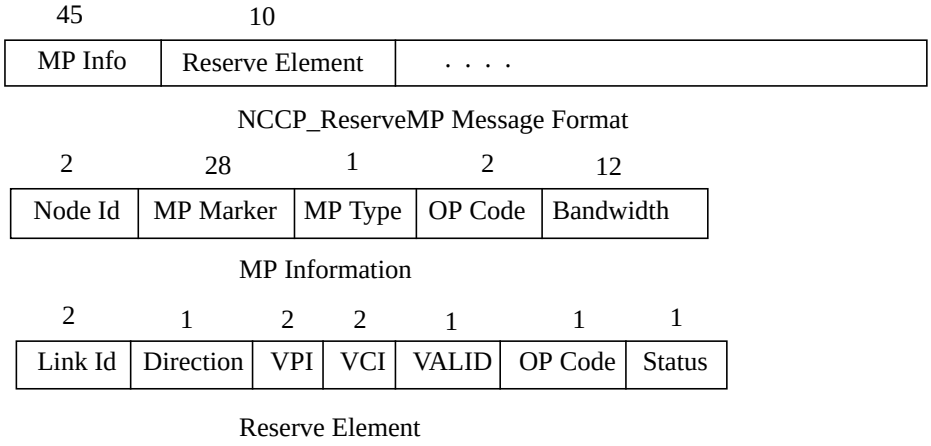

Figure 10. NCCP_ReserveMP Message Format

Multipoint information includes the node identifier (*Node Id);* a rendezvous marker (*MP Marker*) which helps to find the multipoint in the child domain. If the multipoint does not exist yet, this field will be filled with NULL_Marker. *MP Type* takes the value of VP_Connection or VC_Connection. *Op Code* takes a value from Table 5.

Table 5: MP or Sink/Source OP Code

NCMO_OPType	Description
NCMO_OP_NULL	No operation. (There could be operation on sink/ source, but no operation on multipoint)
NCMO_OP_**Alloc**	Allocate a new MP or sink/source
NCMO_OP_D**rop**	Drop an existing MP or sink/source
NCMO_OP_C**ommit**	Commit a previous reserved operation

Bandwidth is a structure with three 4 byte unsigned integers representing the peak rate, the average rate and the burst size.

The sink/source information element contains the following fields. A two-byte unsigned integer *link id* identifies a link in the abstract switch. *Direction* is a one-byte unsigned integer taking the value in Table 6. *VPI* and *VCI* are two-byte unsigned integers. *Valid Code* takes a value defined in Table 7. The *Valid* code tells the child process that

Table 6: Direction Code

Direction	Description
SC_LinkDirection_Null (0)	Both directions are invalid
SC_LinkDirection_ToSwitch (1)	Input link
SC_LinkDirection_FromSwitch (2)	Output link

Table 6: Direction Code

Direction	Description
SC_LinkDirection_Bidirectional (3)	Bidirectional link

Table 7: Valid Code

Valid Code	Description
ALLOCATE_FOR_ME	allocate next available VXI
USE_MINE	verify the carried VXI
INVALID	Do not allocate or verify
LEAVE_IT_ALONE	Same as INVALID

who is responsible to allocate the VXI. The *OP Code* field takes a value from Table 5. The *Status* field is only valid in a response message. It takes the value of either NCMO_SUCCESS or NCMO_FAIL.

4.1.4. NCCP_UpdateHardwareMP Message

An UpdateHardwareMP message tells the child process to set up the hardware tables in physical switches. The *OP Code* field always has the value *NCMO_Commit*. Other fields are the same as in the NCCP_ReserveMP message.

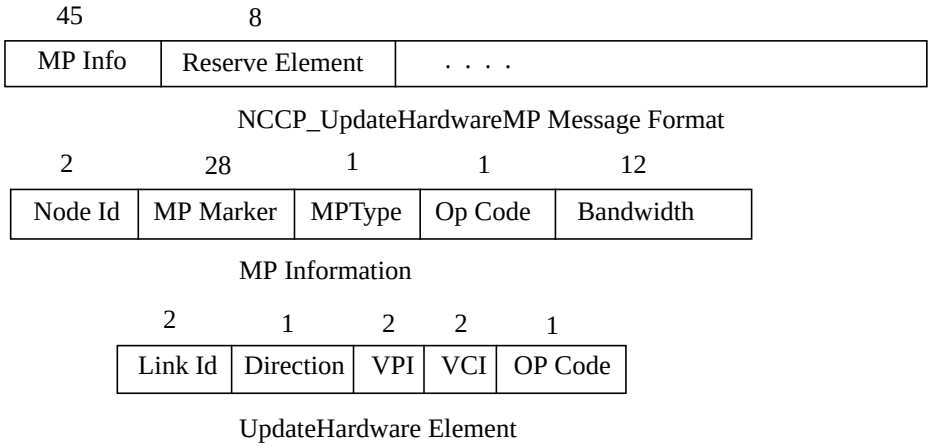

Figure 11. NCCP_UpdateHardwareMP Message Format

4.1.5. NCCP_RollbackMP Message

The higher level process can send a number of reserveMP requests. If one of them fails, it may decide to release

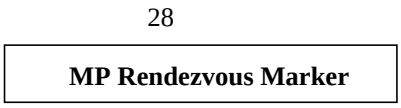

Figure 12. NCCP_RollbackMP Message Format

all the resources reserved for the set of operations. In this case, the higher level process can send an NCCP_RollbackMP request. When an NCCP_RollbackMP request is received, the lower level process should pull the system back to the previously committed state. There is only one information element in the rollback request, a multipoint rendezvous marker, which is used to locate the multipoint on which the operation is performed.

4.2. NCCP Objects

The NCCP API is object-based. There are five main types of objects: the *global NCCP* object, *channel address* objects, *requester* objects, *requestee* objects, and *response* objects. The *global NCCP* object, discussed briefly in Section 4.3, is a per-process object contained in the NCCP library that manages NCCP communications for the process. *Channel address* objects are used to specify the destination of a message. A *requester* object represents a request operation in the process that sends the request message, while a *requestee* object represents the request operation in the process that receives the message; both objects represent request messages. A *response* object represents a response message.

The Core NCCP (C-NCCP) provides the global NCCP class and *base* classes for the requester, requestee, and response objects and the classes that support C-NCCP operations. The developer of an Extended NCCP (E-NCCP) protocol — for example, the designer of Jammer [3] — is responsible for *deriving* additional requester, requestee, and response classes from the base classes. For example, to support PING, the developer might derive NCCP_RequesterPing, NCCP_RequesteePing, and NCCP_ResponsePing classes. The class hierarchy that results is shown in Figure 13. This figure also shows that the NCCP requester base class is derived from the REND_base_class and SCHEDULER_base_class [7].

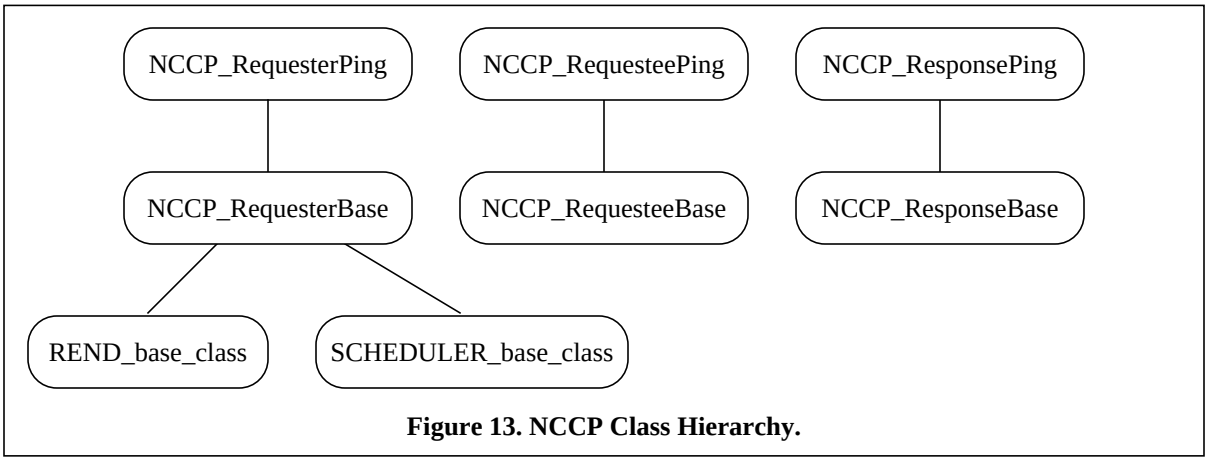

Figure 13. NCCP Class Hierarchy.

4.3. NCCP_GlobalClass and the NCCP_Global Object

The NCCP_GlobalClass is the “engine” of NCCP and manages communications for the protocol. The class contains a single static instance of the class, the NCCP_Global object. A program using NCCP should use only this

object in its interactions.

The NCCP_Global object API provides four types of operation:

1. Channel registration. The *setBTPAgent()* function is used to register the BTPWrapper, the object which is to be used for BTP communications. The *addPriquePair()* function is used to register a PriquePair. The NCCP may have only one BTPWrapper, but may have several PriquePairs. It is not possible to unregister a PriquePair after it has been registered. It is possible to change the BTPWrapper or unregister it (by registering a NULL pointer), but the utility of this is questionable.

2. Scheduler registration. The *setScheduler()* function is used to register a Scheduler_class object [7] with NCCP. This object is then used by NCCP to cause time-outs on requester objects. It is not necessary to register a scheduler with NCCP; however, if a scheduler is not registered it is not possible to schedule time-outs on requester objects.

3. Message sending. The *send()* function takes as arguments an NCCP_ChannelAddress (see Section 4.4) and a ByteBuffer containing a message. The function transmits the message over the selected channel. This is a private function which is available only through the *send()* functions of message base classes.

4. Protocol engine. The *touch()* function examines each of the registered communications channels to determine if a message is available. If a message is available, it is read and processed. Note that when each of the communications channels is examined, if no message is available on any of the channels, the context switcher will be called [7]. The *touch()* function also calls the scheduler, if one has been registered, thereby causing time-outs on request objects or other scheduled time-outs.

The typical use of NCCP is thus relatively simple. The process first opens the communications channels that will be used (in the case of the GBNSC, this is the PriquePair to its CMSS parent and a BTPWrapper for accessing the ATM card and Jammer) and registers them with the NCCP_Global object. A scheduler is then created and registered. In the main loop (or equivalent construct) of the process, the *touch()* function is called to process any messages that have arrived and check for time-outs. When the process wishes to initiate an NCCP operation, it creates a requester object of the appropriate derived type and calls that object's *send()* function to cause a message to be transmitted. Responses are handled in a similar way; a response object is created and its *send()* function is called.

4.4. NCCP_ChannelAddress

The NCCP_ChannelAddress class is used to represent the destination of an NCCP message. Objects of this type represent either a particular PriquePair or a BTP address. A PriquePair uniquely specifies the destination process since each PriquePair connects to only one other process. A BTP address is used by NCCP to communicate over the registered BTPWrapper, with the address uniquely specifying the destination process.

An object of this type must be supplied to the constructor of the NCCP_RequestBase class; in other words, when a process using NCCP wants to make a request, it must specify the process to which the request is to be sent. The object is copied into the requester object and is used by that object's *send()* function to transmit the request to the target process. When a request message is received, a requestee object is generated which contains the source of the message. This source is then passed on to each response object that is created for the requestee object, and is then used by the response object's *send()* function to transmit the response message back to the source of the original request.

4.5. NCCP_RequesterBase and Requester Objects

A requester object represents an operation request in the process that originated the operation. When a process determines that it must perform an operation, it creates a requester object of the appropriate type and sets the parameters of the object. It then calls the object's *send()* function, which causes a message to be sent to another process. The process may also schedule a time-out for the requester object. The process can then do other work, being sure to call the NCCP_Global object's *touch()* function. As responses to the operation arrive, they are rendezvoused with the

requester object and processed by that object. The object may also time-out, if the response(s) fail to arrive within the scheduled time limit. The object may be deleted at any time; the rendezvous mechanism will take care of discarding any responses that arrive after that time.

The base class for requester objects contains three data members. The operation type is an unsigned integer which distinguishes among all the various operation types in NCCP. The class also contains an NCCP_ChannelAddress, which, as described above, indicates the destination of the request. Finally, the class contains a Register_List_class pointer which is used in scheduling.

The base class constructor must be provided the operation type and the destination address. These are stored in the object, and the other base class data is initialized. The derived class constructor may then set the values of any additional class members. The virtual destructor does not need to do anything special.

As indicated in Figure 13, the NCCP_RequesterBase is derived from two other classes, the REND_base_class and the Scheduler_base_class. The REND_base_class provides a rendezvous mechanism which is used to direct incoming responses to the appropriate requester object. The deriver of a requester class is responsible for defining a virtual *REQ_rendezvous()* function which takes a void pointer argument. This function will be called when a response arrives. The function argument is a pointer to a ByteBuffer containing the response message.

The Scheduler_base_class handles time-outs on requester objects. The deriver of a requester class must define a virtual *timed_out()* function taking no arguments. This function will be called when the requester object times out. All *timed_out()* functions should call the *baseTimeout()* function of the requester base class. This function "cleans up" after the time-out so that the destruction of the requester object is performed correctly.

Support for time-outs is provided through two base class functions. The *scheduleTimeout()* function takes two arguments representing seconds and microseconds (the latter is, by default, 0) and, using the scheduler registered with the NCCP_Global object, sets up a time-out after the indicated interval. The *cancelTimeout()* function cancels any previously-scheduled time-out.

The base class *send()* function is used to transmit the request. The function makes use of the virtual *store()* function, which places the contents of the request into a ByteBuffer. If an operation does not have any data that must be transmitted (for example, a PING would probably not need to pass any data to the other process), this function need not be defined. However, if the operation needs to pass data (for example, a ReserveMP), a *store()* function for the requester object must be defined. The first thing that the *store()* function should do is to call the base class *storeHeader()* function, which places a message phase (a constant, in this case representing a request message), the operation type, and the requester object's rendezvous marker into the ByteBuffer. The *store()* function may then add other data to the ByteBuffer. The routine should add no more than 950 bytes of additional data, a limit related to the maximum message size in PRIQUE and BTP communications. {Why does this limit exist?}

4.6. NCCP_RequesteeBase and Requestee Objects

A requestee object represents an operation in the process that received the request message. When a request message is received during a call to the NCCP_Global object's *touch()* function, it is passed to a function *baseHandle()* which consists primarily of a large switch statement. This function examines the message to determine the operation type and creates a requestee object of the appropriate type. The *handle()* function of that object is then called. This function does whatever is necessary to process the operation, as defined by the deriver of the class.

The base class for requestee objects contains three data members. The operation type is a small integer, as used in the corresponding requester class. The class also contains an NCCP_ChannelAddress, which indicates the source of the request and the destination of responses. Finally, the class contains a REND_Marker_class object which is obtained from the requester base class. This is used in constructing response objects, so the response can rendezvous with the requester object.

The derived constructor should take (at least) two arguments, the source address and a ByteBuffer containing the message. The constructor should call the base class constructor with the source address as an argument, then use the

retrieve() function to read the message from the ByteBuffer. The derived constructor may then set the values of any additional class members. The virtual destructor does not need to do anything special.

Each requestee class contains a virtual void *retrieve()* function, which takes the contents of the request from a ByteBuffer. This function should reverse the action of the corresponding requester class *store()* function. If an operation does not have any data that must be transmitted in the request message, the *retrieve()* function need not be defined. However, if the operation needs to transmit data, a *retrieve()* function for the requestee object must be defined. The first thing that the *retrieve()* function should do is to call the base class *retrieveHeader()* function, which extracts the message type (request), operation type, and rendezvous marker from the ByteBuffer.

The derived requestee class should define a virtual *handle()* function. This function is called by the *baseHandle()* function after the requestee object is created. It does whatever processing is necessary for the operation. If the function returns TRUE, the requestee object will be deleted by the *baseHandle()* function. This would be used if the process can immediately respond to the request. If it returns FALSE, the requestee object will not be deleted. This would be used if the process wants to keep the requestee object (*e.g.*, on a list) and process it later.

4.7. NCCP_ResponseBase and Response Objects

A response object represents a response message. The same object type is used in both the requester process and the requestee process; however, the objects are only generated in the requestee process. When the requestee process determines that it should send a response to a particular requester object, it creates a response object of the appropriate type and sets the parameters of the object from the requestee object. These parameters include the message destination and rendezvous marker. It then calls the response object's *send()* function, which causes a message to be sent to another process, and deletes the object. The message is received by the other process in a call to the NCCP_Global object's *touch()* function. The rendezvous marker is extracted from the message and used to rendezvous with the original object. A pointer to a ByteBuffer containing the response is passed via the rendezvous.

The base class for response objects contains four data members. The operation type is a small integer and is the same as the operation type for the requester and requestee objects. The NCCP_ChannelAddress indicates the destination of the response. The REND_Marker_class object is used for rendezvous. Finally, the class contains a Boolean end-of-message marker which may be used by the derived classes when the response to a request must be broken up into several messages.

Response classes should have two constructors. The first constructor, used on the requester side, should take as its argument a ByteBuffer reference and should first call the *retrieve()* function with the same argument. This function initializes the response object from the message contained in the ByteBuffer. The second constructor, used on the requestee side, should take as one of its arguments a requestee object reference and invoke the base class constructor with the same argument, cast to a reference to a requestee base class. The base class constructor will initialize the operation type, channel address, rendezvous marker, and end-of-message marker, after which the derived class constructor can initialize any additional data as desired. Figure 13 in Section 5.7 gives an example of the two constructors. The virtual destructor does not need to do anything special.

The derived response class should define virtual *store()* and *retrieve()* functions to place the contents of the object into, or take them out of, a ByteBuffer. The two functions should reverse one another's actions. If the response does not contain any data, these functions may be omitted. If they are defined, the first thing that the *store()* function should do is to call the base class *storeHeader()* function; and similarly the first thing that the *retrieve()* function should do is to call the base class *retrieveHeader()* function.

5. The NCMO

The NCMO (Node Controller Managed Object) represents the interface between a CMSS parent and a child process. Each parent process in the CMSS has an NCMO object for each of its children, as shown in Figure 14. The NCMO provides an API which allows creation of the child process and provides connection-oriented operations on an abstract switch.

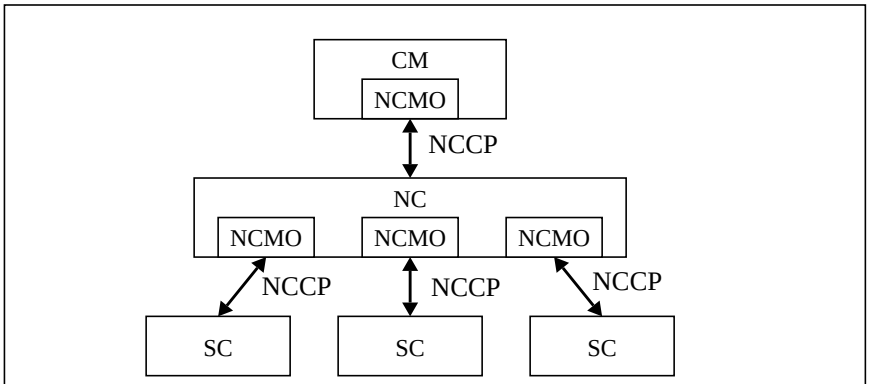

Figure 14. NCMOs in the CMSS Tree.

The operations on an NCMO are transmitted from the parent to the child process through NCCP messages. The child process translates the abstract operations into more concrete operations, as appropriate to the particular process. For example, if the child is an NC, it will translate the NCMO operations into operations on its own NCMO objects; if the child is a GBNSC, it may translate the NCMO operations into control cell sequences which construct and modify cell recycling distribution trees; and if the child is some other type of SC, it will translate the NCMO operations into whatever actions are appropriate to implement the commands.

The remainder of this section first presents the processing that is performed when an NCMO object is initialized. Then we introduce NCMO objects followed by the discussion of using the NCMO's reserve/commit/update protocol. The final portion of this section presents some aspects of the implementation of the NCMO.

5.1. NCMO Initialization

The *initialize()* function for an NCMO object currently takes eleven arguments:

1. An integer pair of (*Node Id*, *Nod Id mask*) which uniquely identifies the child within the node domain. As described in Section 2, the node construction can be nested. A parent node is responsible for assigning a unique *node id* for each of its children. This local *node id* are appended to the parent *node i*d to form a hierarchical *node id*. The *node id mask* indicates the number of significant bits in the node id.

2. An *NCMO Id*, which is an integer chosen by the parent process. This integer is not processed by the child in any way, but is included in the NCCP messages from the child so the NCCP requester or requestee objects can determine which NCMO object is affected by the message. The user of the NCMO must provide a function *mapIdentifierToNCMO()* which takes an identifying integer and returns a pointer to the appropriate NCMO object.

3. A string containing the name of the configuration file that the child process is to use.

4. A pointer to a Signal Handler object [7]. This will be used in creating the PriquePair communications channel.

5. A pointer to a Context Switcher object [7], also used in creating the PriquePair.

6. A rendezvous marker object indicating whom to report to when link states change.

7. A pointer to a Scheduler object [7] which is to be used for time-outs during the process creation. This argument defaults to a NULL pointer, indicating that no time-outs will occur.

8. An integer value indicating the time in seconds that the process will wait for the child to initialize itself.

The default value is 30.

9. An integer *simulation slot* number, used to contact the network or network simulator. This value defaults to 0 (the "real" network).

10. An integer *debug leve*l which will be used to determine how much debug information to be printed. The default value is 0 (no printing).

The *initialize()* function first creates a new PriquePair object using the *signal handler* and *context switcher* provided. The parent process registers the PriquePair object with the NCCP global object. A child process is started. If a scheduler object was provided, the parent process schedules a time-out. The parent process then enters a loop in which it touches the NCCP global object until the child process responds with a ChildStatus message, the child process dies (a SIGCHLD signal is caught), or a time-out occurs. The parent *initialize()* function then returns. The caller may inspect the child status at that time to see if the child is ready or if its initialization failed.

The initialization function in the forked child process creates a command line for the subsidiary:

<executable> ***nodeId nodeIdMask ncmoNumber configFile priqueId1 priqueId2 simSlot debugLeve*****l**

where the *executable* is the program to run, *childId*, *childIdMask*, *ncmoNumber*, *configFile*, *simSlot*, and *debugLevel* arguments are the ones passed to the *initialize()* function and *priqueId1* and *priqueId2* are the shared-memory identifiers for the PriquePair communications channel. The subsidiary process initialize itself withe the configuration file provided. At the end of its initialization, it sends an NCCP ChildStatus message to report its state.

5.1.1. NCMO Manager Library

The NCMO Manager library provides a convenient means to manage NCMO objects, particularly for a Node Controller process which have many such objects. The library provides a static instance of an NCMO manager object. Internally, the manager maintains a table of NCMO objects which are identified by index (starting with 0). The signal handler, context switcher, scheduler, default time-out, simulation slot, and debug level parameters of an NCMO object are also contained in the manager; the same values are used for all the NCMO objects. The user of the manager object must first define the number of NCMO objects that are to be used and provide the signal handler, context switcher, and other global parameters.

The process may separately initialize each of the NCMO objects by providing the index, the configuration file, and (optionally) a time-out value to override the default value. Deleting the NCMO manager object causes the deletion and shutdown of all the subsidiary processes. The NCMO manager object provides a boolean *baseHandle()* function which takes an NCCP operation type, NCCP channel address, and ByteBuffer. If the message in the ByteBuffer corresponds to one of the NCMO messages, the function will process it (create the requestee object and call its *handle()* function) and return TRUE. Otherwise it will return FALSE. This function can be used in the default case of the NCCP requestee base class *baseHandle()* function which the programmer must supply.

The NCMO manager library supplies the *mapIdentifierToNCMO()* function that is required by the NCCP operations. This function may also be used by the process to obtain a pointer to the NCMO object, which may be used to access all the operations on the NCMO.

5.2. NCMO Objects

Table 15 shows all the NCMO objects and their inheritance relationship.

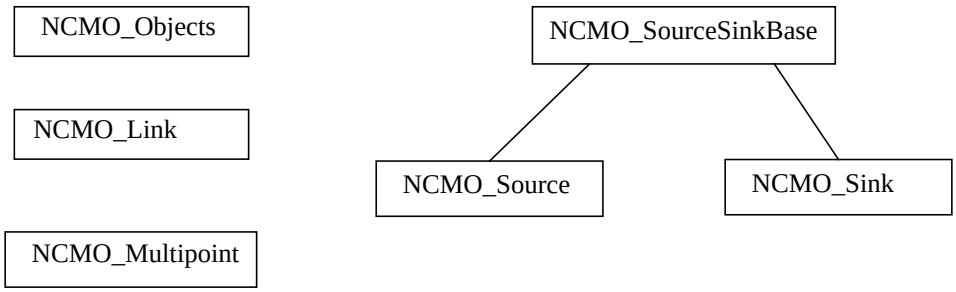

Figure 15. NCMO Objects

5.2.1. NCMO_Object

An NCMO object holds pointers to some commonly accessible objects such as *context-switcher, scheduler.* It is responsible for generating MultiPoint (MP) objects as required. After the creation of MPs, the responsibility to maintain these MPs is left to the user who requested the creation of the MPs. An NCMO object maintains a list of NCMO_Link objects. All the information for a link comes from the NCCP_LinkStatus message and is set by a *discloseLink()* member function.

5.2.2. NCMO_Link

An NCMO_Link object keeps all the attributes of a link and the link status. An NCMO_Link object is responsible for generating the NCMO_Source and NCMO_Sink objects. After creation, it is the user's responsibility to maintain these objects. In the current implementation, the NCMO_Link object does not check the resources or QOS when it is called to generate a sink or source. Part of the resource allocation and QOS checking could be built into the member function of this object.

5.2.3. NCMO_Multipoint

An NCMO_Multipoint object represents a multipoint-to-multipoint connection. Some important fields of an NCMO_Multipoint are listed in Table 8. Table 9 lists the public functions of an NCMO_Multipoint.

Table 8: Attributes of a NCMO_Multipoint

Field Name	Description
type	VP or VC
state and prevState	Keep the current connection state.
bandwidth	Resource requirement for the connection
source and sink list	lists of NCMO_Sources and Sinks
upMarker	rendezvous object to report to higher level
selfMarker	this MP's rendezvous marker passed to the child

Table 8: Attributes of a NCMO_Multipoint

Field Name	Description
downMarker	The rendezvous marker for the MP in child

Table 9: Public Functions of NCMO_Multipoint

Function Name	Description
reserve()	reserve resources for the MP
updateHardware()	send message to subsidiary to set up switch tables
rollback()	bring the MP back to previously committed state
connectSource(NCMO_SourceSinkBase*)	add a source into its source list
connectSink(NCMO_SourceSinkBase*)	add a sink into its sink list
disconnectSource(NCMO_SourceSinkBase*)	remove a source from its source list
disconnectSink(NCMO_SourceSinkBase*)	remove a sink from its sink list
REQ_rendezvous(void*)	process response operation

5.2.4. NCMO Sink and Source

An NCMO_Sink or an NCMO_Source is derived from NCMO_SourceSinkBase. An NCMO_SourceSinkBase object represents an input or output port for a connection. A NCMO_SourceSinkBase object maintains the following attributes. An NCMO_Sink or an NCMO_Source does not have extra attributes. It just implements the virtual func-

Table 10: Sink and Source Attributes

Field Name	Description
state and prevState	Keep state and for rollback
linkId	Indicating on which link
vpi, vci	the VPI, VCI value
mp	a pointer to the multipoint

tions defined in NCMO_SourceSinkBase. Following are the most important member functions.

Table 11: Public functions of Sink and Source

Function Name	Description
alloc(VPI, VCI, ValidCode)	synchronously set parameters.
drop()	set state and add to MP's reserve list
commit()	set state and add to MP's commit list
rollback()	return to previously committed state
packReserveMessage(list)	add this element info into the list
packUpdateMessage(list)	add this element info into the list
unpackReserveMessage(NCCP_ResponseStatus, NCCP_ReserveElement*)	set state based on the values in the reserveElement.
unpackUpdateMessage(NCCP_ResponseStatus, NCCP_UpdateHardwareElement*)	set state based on the values in the updateHardwareElement
connectMP(NCMO_Multipoint*)	add itself into MP's source (sink) list
disconnectMP()	remove itself from the MP's source (sink) list.

5.3. NCMO Reserve/Commit/Update Protocol

The discussion in the previous section may have given the impression that manipulating the NCMO objects (creating sources/sinks/multipoints and connecting them together) immediately causes changes in the switch hardware so that the connections are physically realized. This is emphatically *not* the case. To build a connection structure, the CM (or CMSS parent) must perform three distinct steps in order to affect the hardware and realize a connection:

1. Reserve - guarantees that the requested resources are available. The reserve operation may fail due to unavailable resources. The NCMO guarantees that once objects have been reserved, the remaining two steps can be performed (i.e., the switch tables can be updated to realize the connections) without failure.

2. Commit - indicates which objects and connections should be realized.

3. Update - changes the hardware tables so the connections represented by committed objects are realized.

There are thus four classes of NCMO functions: the object manipulation (creation, parameter-setting, and connection) functions; the reserve functions; the commit functions; and the update functions. The object manipulation and commit functions are synchronous, in that when the function returns the operation is complete. These synchronous functions are entirely local to the NCMO, with no communication with any subsidiary process.

The reserve and update functions are asynchronous, in that the function initiates the operation and returns, and the operation completes some time later. The asynchronous functions send messages to the subsidiary processes using NCCP. The operation is completed when the NCCP response(s) are received. The asynchronous functions are passed

a rendezvous marker object when they are called. The NCCP operation, when it receives the last response message, will use this rendezvous marker to communicate the results of the operation to the initiating entity.

Figure 16 shows sample code for an operation that creates a VC connection between two terminals connected to a particular node. The code first creates source, sink, and multipoint objects, sets their parameters (the VPI and VCI are set in the object-creation calls), and connects them. All of these function calls are synchronous and merely create data structures in the NCMO object.

```
mp = ncmo->newVPMP(&myMarker);
mp->setBandwidth(...);

vcSink = ncmoLink->getSink(VC_Connection, 3,0, USE_MINE);
vcSource = ncmoLink->getSource(VC_Connection, 0,0, ALLOCATE_FOR_ME);

vcSink->connectToMP(mp);  vcSource->connectToMP(mp);
vcSink->reserve(); vcSource->reserve();
mp->reserve();

vcSource->commit();
vcSink->commit();

mp->updateHardware();
```

Figure 16. Sample Code

The NCMO then reserves the multipoint. This function creates an NCCP requester object representing an operation which reserves the resources used by the multipoint and by all the sources and sinks connected to the multipoint (it is also possible to separately reserve the sources and sinks) and sends the request to its subsidiary. (Probably what is needed here is for the multipoint object to be updated to agree with the state of the connected sinks and sources and then to have that state passed down in the NCCP request.This is the first indication to the subsidiary that there are sources and sinks so the information in the request will have to be sufficient to create the necessary objects in the subsidiary.) When the function returns, the CM can go on to do other processing while it waits for the asynchronous operation to complete. However, it must touch the NCCP global object periodically, to cause processing of the NCCP responses. This break in the sequence is marked by the first wavy line in the figure. At some later point, the NCCP operation will complete and a rendezvous will occur at the requester. The rendezvous will pass information about the operation — minimally, whether or not it succeeded. If it succeeded, the CM can continue with the code shown below the first wavy line.

The next step is to commit the source and sink. These are synchronous functions and only modify the data structures in the NCMO object (marking the objects as committed).

The final operation is to call the *updateHardware()* function of the multipoint. (The suffix -Hardware is used to emphasize that this operation, and this operation only, modifies the physical switches.) This function first marks the multipoint object as committed (if it was not already committed) — but it does *not* commit any of the sources and sinks connected to the multipoint. It then creates an NCCP_UpdateHardwareMP requester object for the operation that commits and updates all the changes to the multipoint and to the sources and sinks connected to the multipoint and sends the request. When the function returns, the CM may do other processing. When the NCCP operation completes, a rendezvous occurs at the requester. Assuming all the switches are still running, the updateHardware operation should always succeeds (the reserve guarantees this) so the rendezvous serves mostly to indicate that the operation has completed.

6. Conclusion

We have described in some detail the design and implementation of the Node Control Communications Protocol (NCCP) and the Node Control Managed Object API. These two software libraries allow for the development of Connection Management functionality on an abstract switch model, thus facilitating the implementation of Connection Management across a network comprised of heterogenous switches.

References

[1] ANSI T1S1 Technical Sub-Committee. Broadband Aspects of ISDN Baseline Document. T1S1.5/90-001, June 1990.

[2] ATM Forum, “The ATM Forum Technical Committee User-Network Interface (UNI) Specification Version 3.1”, The ATM Forum 1994.

[3] O.M. Beal, “Jammer Language Description: A Script Language for GigaBit Switch Testing,” Washington University, Applied Research Laboratory Working Note ARL-96-01, March 1996.

[4] CCITT. Recommendations Drafted by Working Party XVIII/8 (General B-ISDN Aspects) to be Approved in 1992, Study Group XVIII—Report R 34, December 1991.

[5] CCITT Recommendation Q.931 (I.451), ISDN User-Network Interface Layer 3 Specification, Geneva, 1985.

[6] K. Cox and J. DeHart. “Connection Management Access Protocol (CMAP) Specification,” Washington University, Department of Computer Science Technical Report WUCS-94-21, Version 3.0, July 1994.

[7] J. DeHart, “Connection Management Software System (CMSS) Architecture,” Washington University, Applied Research Laboratory Working Note ARL-95-03, June 1996.

[8] J. DeHart and D. Wu, “Connection Management Network Protocol (CMNP) Specification,” Washington University, Applied Research Laboratory Working Note ARL-94-14, Version 1.0 DRAFT, September 1994.

[9] J.S. Turner, “A Gigabit Local ATM Testbed for Multimedia Application.” Washington University, Applied Research Laboratory Technical Report ARL-94-11 Version 3.1, January 1996.

[10] D. Wu, K. Cox, and J. DeHart, “GBNSC: The GigaBit Network Switch Controller,” Washington University, Applied Research Laboratory Working Note ARL-94-12, Version 1.2, June 1996.