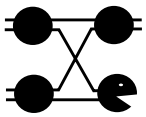


GBNSC:
The GigaBit Network Switch Controller

Dakang Wu
John D. DeHart
Kenneth C. Cox

Version 1.3
Applied Research Laboratory
Department of Computer Science
Washington University
St. Louis, Missouri 63130
jdd@arl.wustl.edu

Working Note ARL-94-12
July 6, 1998

Abstract

The GigaBit Network Switch Controller (GBNSC) is a process which controls the Washington University Gigabit Switch (WUGS) using in-band ATM control cells. The GBNSC is being developed by the Applied Research Laboratory (ARL) of Washington University in St. Louis. This document describes the design and implementation of the GBNSC and is intended for use by those interested in the functionality of the switch controller as well as developers who may be called on to update or maintain it.

Version Notes:

Version 1.0: All Ken Cox

Version 1.1: John's additions. More NCMO details.

Version 1.2: Dakang Wu's addition: 1) separate of GBNSC and NCMO, NCCP

2) GBNSC implementation structure

Version 1.3: John adds modifications for new communication structures.

1. Introduction

Because of the diversity of ATM switching equipment available, we have designed our control software with a number of abstractions in order to encapsulate and conceal that diversity. Figure 1 illustrates a number of these control abstractions. A *node* abstracts a switch or collection of 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 (CMSS)[13].

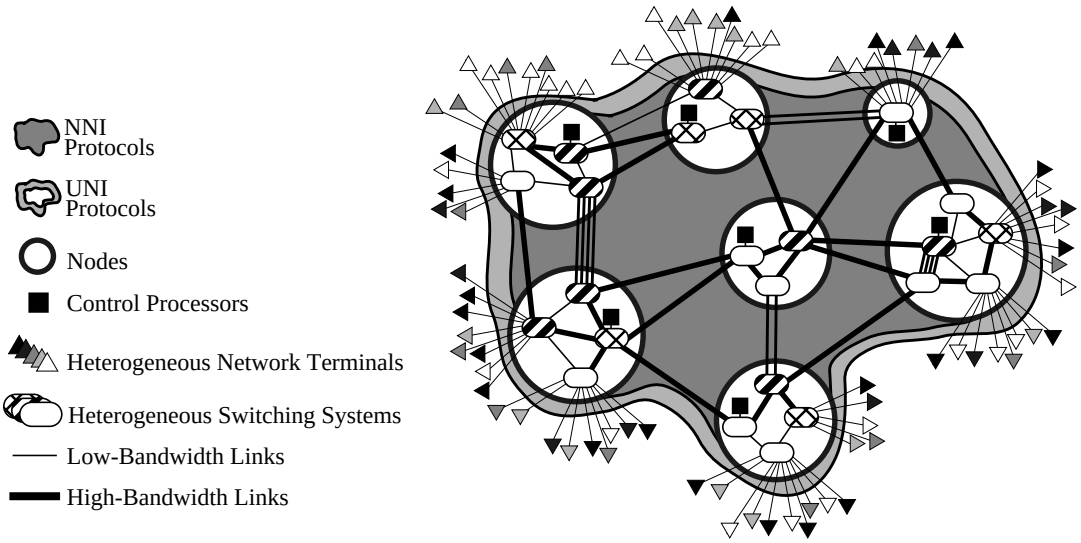


Figure 1. Control Abstractions for ATM Networks.

The Core CMSS present at each node is structured in three layers as shown in Figure 2. The *Connection Management Layer* is distributed across all the nodes of the network, and uses the NNI protocols of Figure 1 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 are presented as a single large switch to the Connection Management Layer. The Node Management Layer 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. The UNI protocols are implemented by a fourth layer, the *Session Management Layer*, which is not considered part of the core CMSS. This layer is only necessary at nodes which support links to clients. It abstracts the whole network and pre-

sents a unified view of it to the clients.

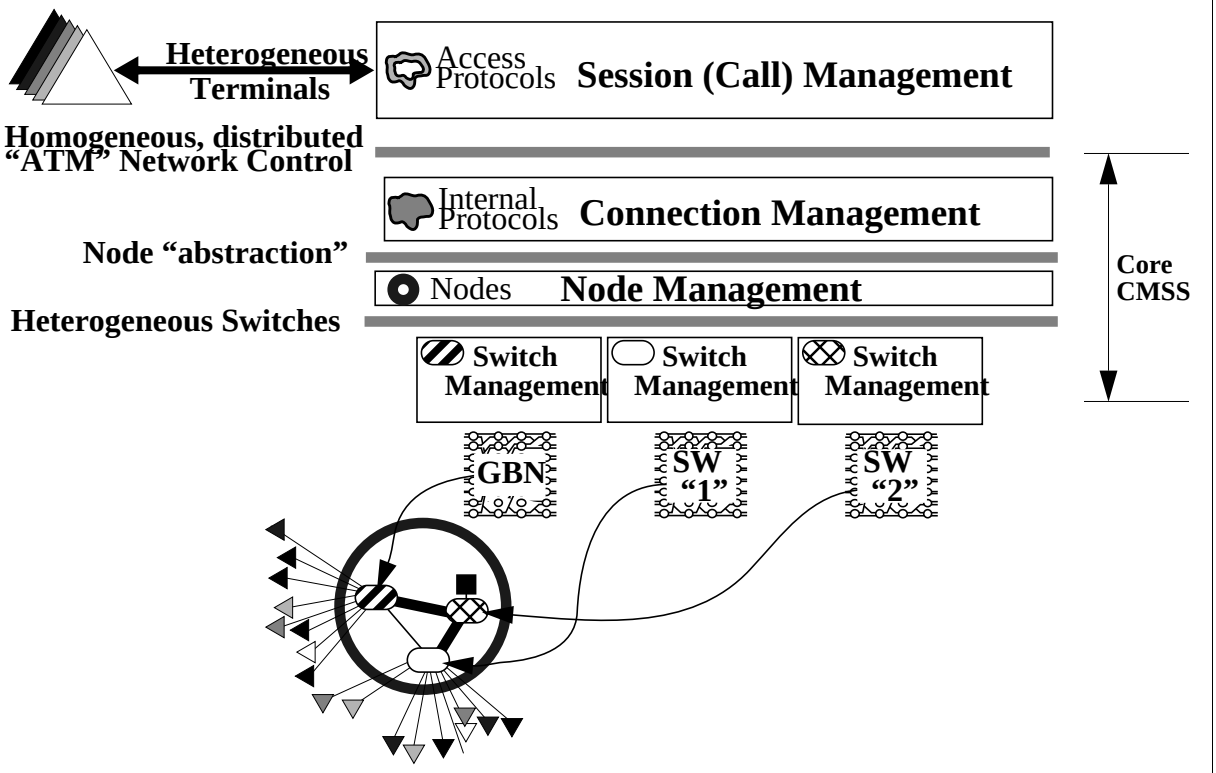


Figure 2. Core Connection Management Software System Plus Session Management

Another software entity, Jammer [4], has been designed and implemented. Jammer provides a flexible script language for manipulating and testing the prototype switches designed and built at Washington University. Jammer does not fit into the CMSS but is a separate entity that communicates directly with a Switch Management module.

This document describes the design and implementation of one specific Switch Management layer which supports the Washington University Gigabit Switch (WUGS) [27]. The GigaBit Network Switch Controller (GBNSC) encapsulates the particular hardware details of the WUGS and provides the proper communications mechanisms for controlling that switch.

Section 2 lays out the top level design of the GBNSC by describing its major components. Section 3 gives the details of how the GBNSC communicates with other entities. The operations of the GBNSC and how they are performed are described in Section 4. The ATM cell interface to the WUGS is detailed in Section 5. Section 6 gives the details of the initialization process for the WUGS. The operations concerning connection building and manipulation are described in Section 7.

2. Major Components of the GBNSC

Figure 3 shows the main objects contained within the GBNSC. These objects are described briefly below (later sections of this document provide greater detail). After the brief descriptions, the processing performed by the GBNSC and the role of the major objects in this "main loop" are described at a relatively high level,.

GBNSC_SwitchController. The global switch controller object is the center of the system. It is responsible for

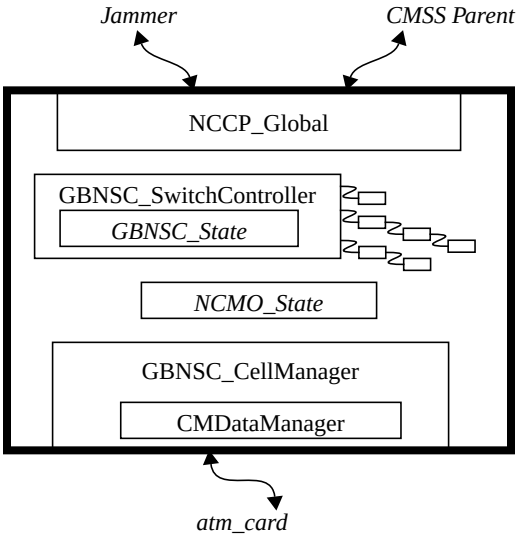


Figure 3. GBNSC Components.

switch initialization and for keeping track of GBNSC requestee operations (Section 4) and of the current hardware state. Initialization is performed using a configuration file as described in Section 6.1. Operations are kept on three lists. The to-be-initiated list contains operations which have not been started. The in-progress list contains active operations (e.g., operations that are waiting for a cell to return from the switch). The complete list contains operations that have finished and are ready to be deleted. The current hardware state is represented by tables in link, IPP and OPP objects within the switchController object. This state information includes the values of all the maintenance registers and VXT entries in the switch. (NB: This appears to be not completely true. I think this state gets initialized at boot time but is not maintained over time. JDD)

NCCP_Global. The NCCP global object is responsible for communications with the CMSS parent and with Jammer. This communication is message-based, using a request-response protocol, the Node Control Communications Protocol (NCCP) [29]. As a simple example, Jammer might wish to read a VCXT entry from the switch. Jammer would create an NCCP requester object and send the corresponding message to the GBNSC. The NCCP global object, on receiving this request message, would create an GBNSC requestee object (Section 4.2.2) and call the handle() function of the object to start the operation. The function would send a RDVCXT cell to the switch. On its return, the cell would rendezvous with the requestee object, which would then create an NCCP response object and send the corresponding message back to Jammer. This message would rendezvous with the original requester object, completing the NCCP operation.

GBNSC_CellManager. The cell manager (Section 5) is responsible for communications with the switch. It reserves the control connections that the switch controller will use, sends cells to the switch, and receives cells from the switch. It uses the CMDATA manager to rendezvous returning cells with the appropriate operation.

NCMO_State. The NCMO state is the connection-oriented view of the switch that is provided to the CMSS parent. Operations on NCMO objects (Section 7) are translated into operations which modify the switch hardware and update the state information in the switch controller object.

2.1. GBNSC Main Loop

The GBNSC is event-driven, with the events of interest being the receipt of an NCCP message from its CMSS parent or Jammer, the receipt of a cell from the switch controller, and the time-out of an operation. The main loop of the GBNSC services these types of events. The loop does the following:

1. “Touch” the NCCP global object to receive and process any NCCP messages.
2. “Touch” the cell manager object to receive and process any cells from the switch.
3. Remove each operation from the to-be-initiated list and initiate the operation.
4. Remove each operation from the completed list and delete the operation.
5. “Touch” the global scheduler object (part of the GBNSC_SwitchController) to cause time-outs.

This loop continues indefinitely, until the GBNSC is shut down by an external agency. Note that the GBNSC does not examine or manipulate the in-progress list directly. That list may be manipulated as a result of items 1, 2 or 5 above.

3. GBNSC Communications

Figure 4 shows the communications performed by a GBNSC in two ways. The left portion of the figure shows how the GBNSC “talks” to three entities: its parent in the CMSS tree (an NC or CM), the switch, and Jammer. Two protocols are used for this communication. The parent process and Jammer use NCCP, while communication with the switch is in the form of raw ATM cells. The right portion of the figure shows the physical communications channels that are currently used by the GBNSC. The GBNSC potentially has three types of channels, a PriquePair object (a shared memory channel dedicated to communication with another process) [13], a TCPobject (to communicate with any other potentially remote process via TCP sockets) [13] and an ATMCard object which it uses to access a Native ATM socket for communicating to the WUGS.

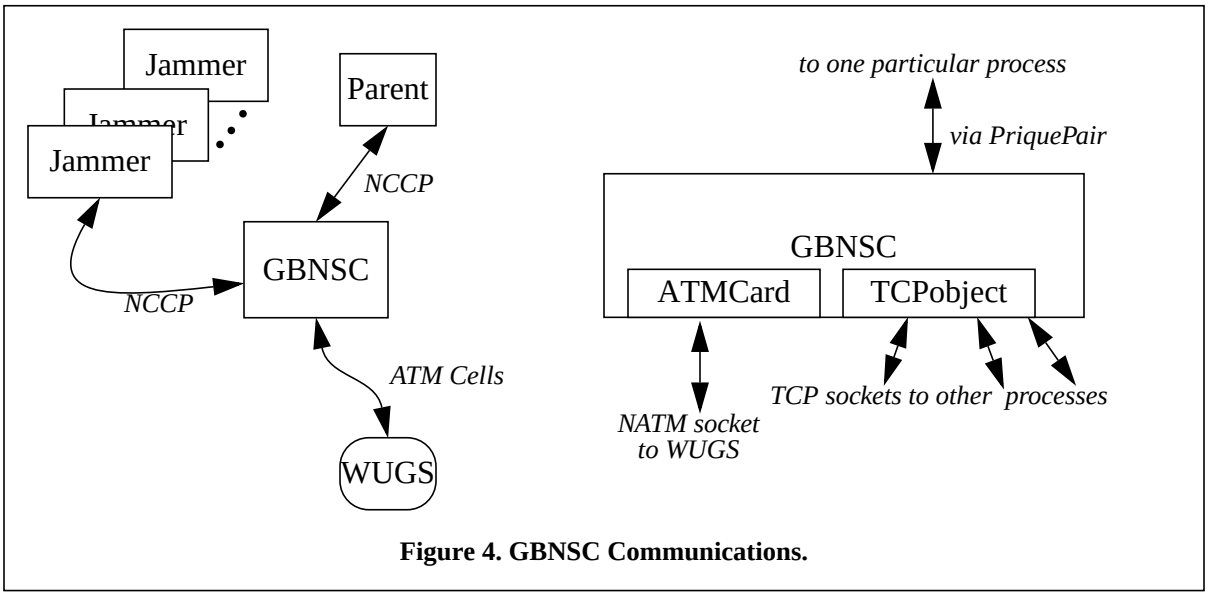


Figure 4. GBNSC Communications.

In the current implementation, the GBNSC uses a TCPObject channel to communicate with its CMSS parent. This channel is established by the GBNSC process when it is started by the NCMO parent. Another TCPObject channel is used to communicate with Jammer and actually allows the GBNSC to be manipulated by multiple Jammers at any given time. A “card manager” object manages the ATM card interface for a GBNSC and allows it to send and receive raw ATM cells.

It is also possible to use a PriquePair channel to communicate between the GBNSC and its NCMO parent. This mechanism, however, is currently being phased out in favor of the more general TCP approach.

3.1. Communications Objects

As the above discussion makes clear, communications in the GBNSC are somewhat complicated. One protocol, NCCP, is split between two channels — a TCPObject or PriquePair channel to the parent, and the TCPObject channel to Jammer. A number of software objects interact to manage the communication and provide a clean structure within the GBNSC. These objects are illustrated in Figure 5.

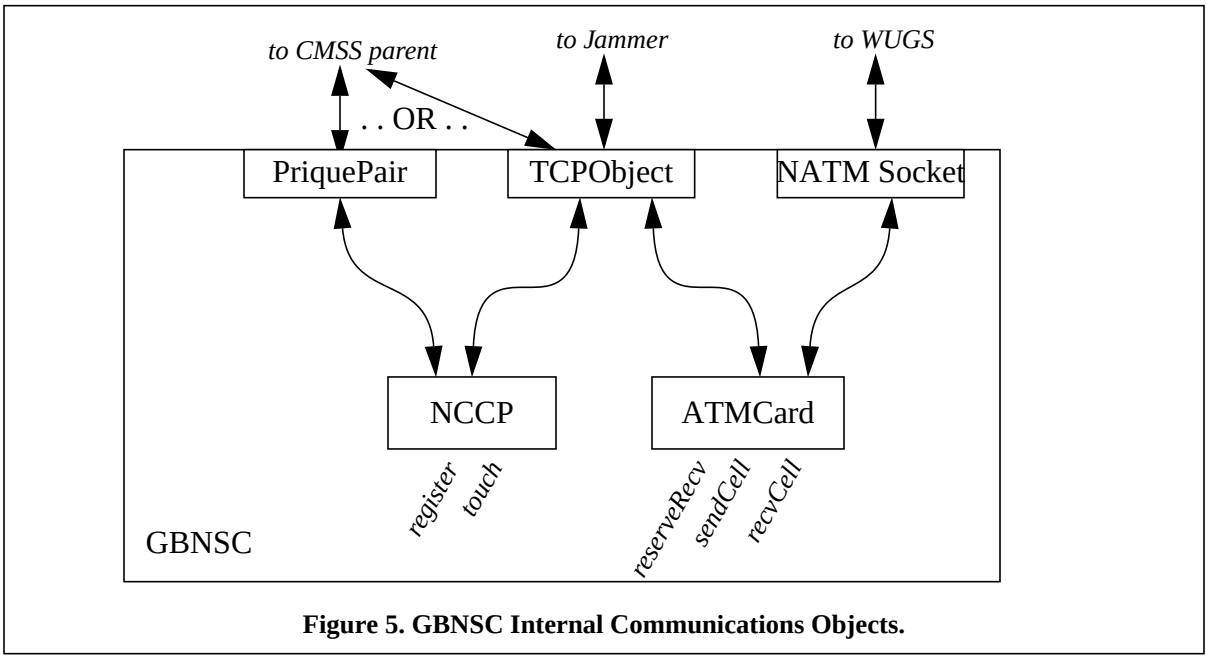


Figure 5. GBNSC Internal Communications Objects.

The TCPObject provides a simple interface for creating and connecting TCP sockets between processes.

The ATMCard object provides an interface to a Native ATM socket. The GBNSC performs function calls on this object, which allow it to manipulate NATM connections to the WUGS. The most important of these functions are the reserveRecv functions and the functions which send and receive cells. The reserveRecvVP and reserveRecvVC functions request that a particular VP or VC be reserved for the process. If the reservation is successful (the requested VP or VC is free), all cells received over the ATM link with the reserved RECV VP or VC will be forwarded to the process. The sendCell function takes 48 bytes of data representing an ATM cell and causes it to be transmitted over the ATM link. The ATM cell header is added by the driver. The recvCell function determines if a cell is available — *i.e.*, was received by the card manager over the ATM link and forwarded to the process. If a cell is available, it is returned by the recvCell function.

The final major component of the GBNSC communications package is the NCCP object, which provides the interface to the NCCP protocol. The object is capable of interfacing to one TCP object and several PriquePair objects (the GBNSC would only uses one PriquePair, but the NC may use several). The NCCP object provides registration functions to define communication channels (TCPObject and PriquePairs) and a scheduler for use by NCCP. The touch function determines if any messages have arrived on any of the registered channels and processes them appropriately. The NCCP is discussed in detail in [29].

Since the GBNSC is largely input-driven, the interface provided to the rest of the GBNSC by the communications package essentially reduces to the NCCP object's touch function and the ATMCard object's recvCell function. By calling touch, the GBNSC causes an NCCP message to be read and processed. By calling recvCell, the GBNSC obtains an ATM cell from the link, which may then be processed.

3.2. Implementation Notes

It is desirable to allow the GBNSC's CMSS parent to be on a different machine. In this case, communications with the parent process would also be over TCP. This is not difficult to accommodate with the existing structure. From the command-line arguments, the GBNSC determines that its parent is remote, and so does not create the PriquePair object or register it with the NCCP object. When it needs to send a message to the parent, it supplies the parent TCP address (obtained from the command line) to the NCCP send function, instead of the PriquePair. The NCCP object then transmits the message using TCP instead of the dedicated PriquePair. [NOTE: This has recently become the preferred means of communication between the GBNSC and its NCMO parent.]

4. GBNSC Operations

GBNSC operations are initiated by two types of events, the receipt of a request message from another process (the CMSS parent or Jammer) and internal events (*e.g.*, time-outs). The operations typically involve sending one or more control cells to the switch and, at some later time, receiving response cells from the switch. If the expected cells do not arrive within the a certain time limit, an error must usually be signalled. Often the operation has some internal state; for example, an operation to update a distribution tree may involve sending one write VXT to the switch and, after the response to that cell has arrived, sending a second write VXT cell. Finally, when the operation is complete the GBNSC may have to send one or more response messages to the process that requested the operation.

4.1. WUGS Operations

Sixteen operations are supported by the WUGS through control cells. The control cell formats are shown in Figure 6. The opcodes which may be used in the control cells are given in Table 1. A full definition of all the fields in the control cells can be found in [27].

The GBNSC uses the recycling nature of the WUGS in order to access the tables at the particular port that is to be updated. To reach the input side of a port it is necessary to recycle through the switch once, as shown in Figure 7a. The control cell format provides three address fields that can be used for routing through the switch. The example of updating an IPP given in Figure 7a uses two of these address fields; one to get to port 3 and the second to get back to port 0 to return to the control processor. Three address fields would be used when we wanted to test a particular path through the switch. The example shown in Figure 7b, shows a way to test the path between port 3 and port 6.

4.2. GBNSC Operation Classes

The class hierarchy shown in Figure 8 is used to support the development of GBNSC operation classes. The hierarchy introduces two new classes. The GBNSC_OperationBase class provides the basic mechanisms for handling operations, as described in Section 4.2.1. Operations that are initiated by GBNSC-internal events are derived from this class. The GBNSC_RequesteeBase class, discussed in Section 4.2.2, is derived from the GBNSC_OperationBase class and also from the NCCP_RequesteeBase class. Operations that are initiated by requests from other processes are derived from this class.

4.2.1. The GBNSC_OperationBase Class

The GBNSC_OperationBase class provides two principle capabilities. The first of these is the handling of time-outs. This is accomplished by deriving the class from the SCHEDULER_base_class as shown in Figure 8. The operation class defines a *scheduleTimeout()* function which schedules a time-out with the GBNSC's global scheduler object. A *cancelTimeout()* function is also provided, as well as a *baseTimeout()* function. The use of these functions is the same as for the NCCP_RequesteeBase class. The deriver of an operation class must provide a virtual *timed_out()* function which will be called when a time-out occurs.

The second capability is support for the rendezvous with returning control cells. Cell rendezvous uses the CMDATA field of the control cell. The operation base class automatically obtains a new CMDATA value when it is allocated and releases the value when it is destroyed. The deriver of a class must provide a virtual *receiveCell()* func-

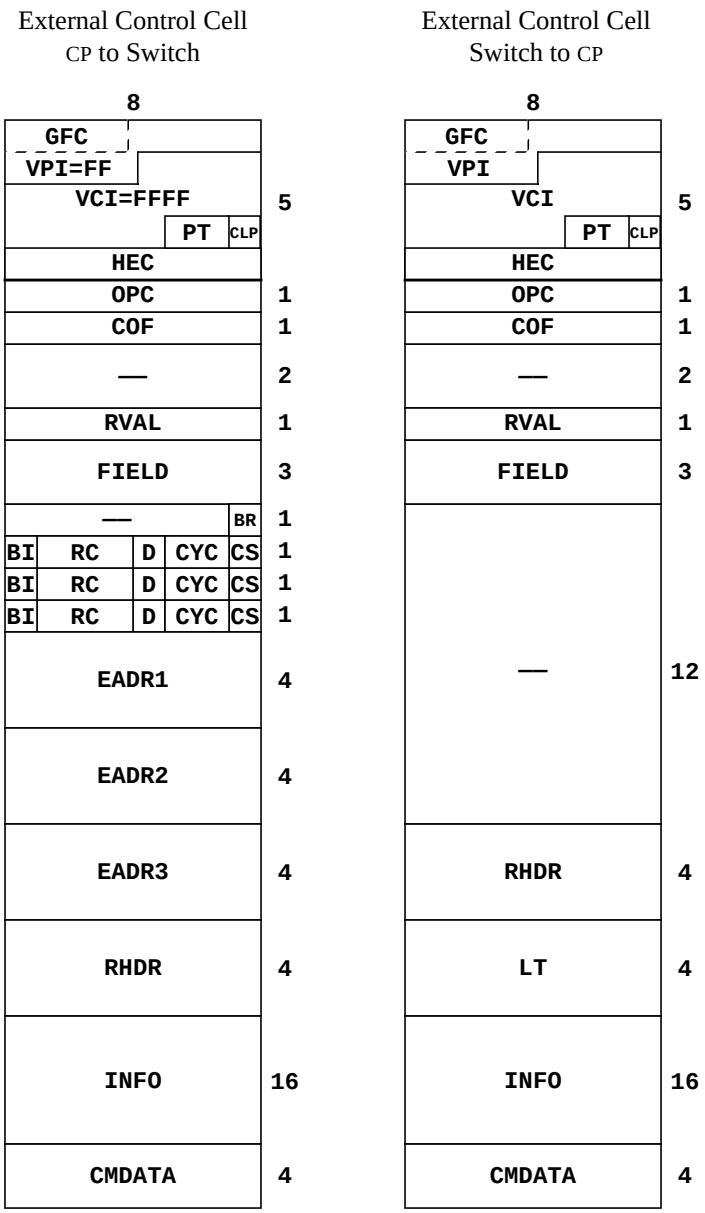


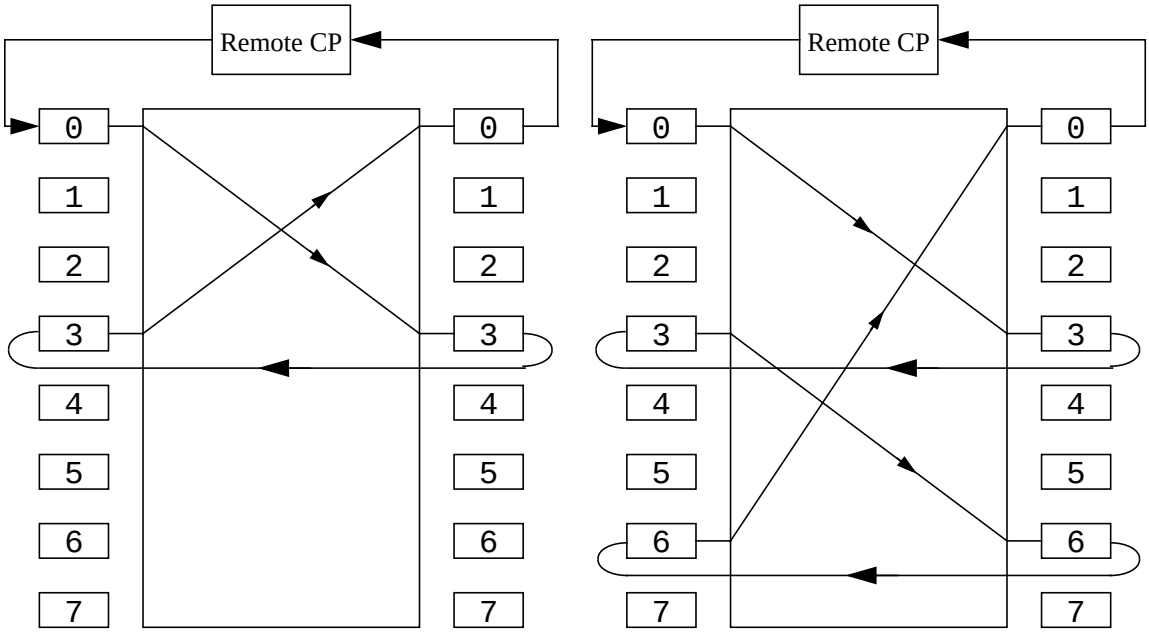
Figure 6. External Control Cell Formats

tion which takes as its argument a control cell received from the switch. The rendezvous mechanism itself is supported by the GBNSC_CMDataManager class, discussed in Section 5.1. The latter class is responsible for allocating and de-allocating CMDATA fields and associating them with operation objects. Using the class, the GBNSC can take a control cell arriving from the switch, look up its CMDATA field to find an associated operation object, and call the *receiveCell()* function of the operation object.

4.2.1.1. Example Use: Ping Protocol

The GBNSC_OperationBase class is illustrated here with the ping protocol. This protocol is to be run periodically to verify that the switch is up and has not performed a hardware reset.

The protocol involves sending a read MR cell to the switch, directing it at IPP 0 and reading the hardware status field. When the cell returns, the reset bit of the returned information is examined to determine if a hardware reset has occurred. If a reset occurs, or if three consecutive ping cells fail to return from the switch, the error is to be reported to



(a) Accessing IPP #3 or OPP #3. (b) Routing from IPP #3 to OPP #6.

Figure 7. Routing a Control Cell

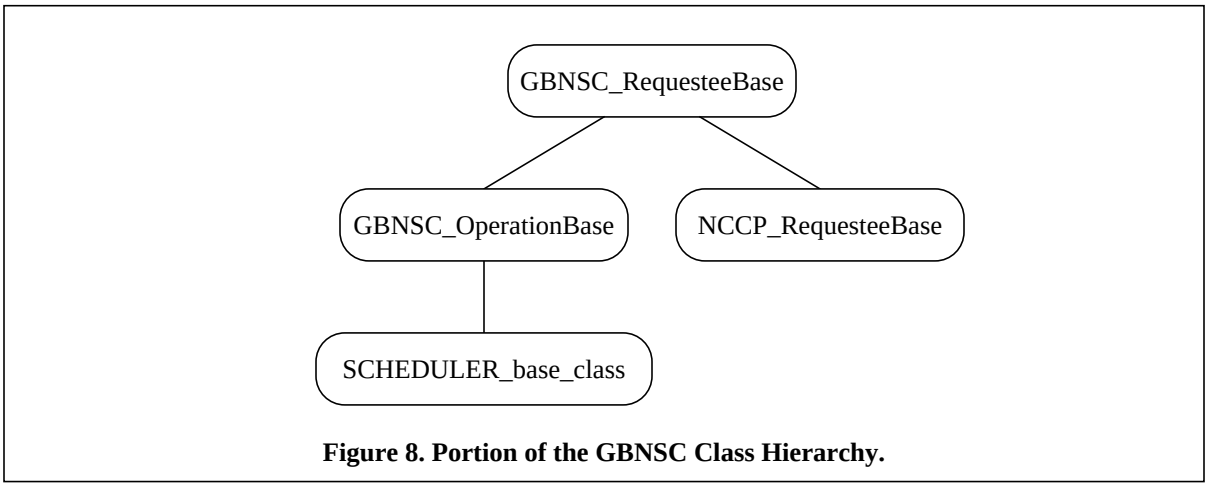


Figure 8. Portion of the GBNSC Class Hierarchy.

the CMSS parent and any Jammer process that is registered. The ping protocol continues to run until it detects that the switch is up again, at which time the switch is re-initialized and the CMSS parent and Jammer are informed. The operation object is permanent; that is, the GBNSC will create a pingProtocol object when it starts, and that object will remain in the system forever, periodically pinging the switch. Most of the GBNSC internal operations are constructed in this manner. (Jammer is not currently informed. JDD)

{What are the other internal operations? Perhaps a list of them would be good.}

Figure 9 shows the definition of the PingProtocol class. The class contains three data members. The Boolean pingRecieved indicates whether the response cell was received, while resetOccurred indicates whether that cell showed that a reset had occurred. The counter missedPings keeps track of how many pings were lost.

The object constructor calls the base operation object constructor with the constant GBNSC_CMDATA_PING.

```
class GBNSC_PingProtocol: public GBNSC_OperationBase {

   private:
      Boolean pingReceived;
      Boolean resetOccurred;
      un2byte missedPings;

   public:
      GBNSC_PingProtocol() : GBNSC_OperationBase(GBNSC_CMDATA_PING) {
         pingReceived = TRUE;
         resetOccurred = FALSE;
         missedPings = 0;
      }
      ~GBNSC_PingProtocol() {}

      virtual void receiveCell(GBNSC_ReceiveControlCell & cell);
      virtual int timed_out();
};


void GBNSC_PingProtocol::receiveCell(GBNSC_ReceiveControlCell & cell) {
   pingReceived = TRUE;
   missedPings = 0;
   resetOccurred = cell.getMR().getReset();
}

int GBNSC_PingProtocol::timed_out() {
GBNSC_TransmitControlCell cell;

   if (!pingReceived) {
      missedPings++;
   }
   if (resetOccurred ||
       (GBNSC_State.getStatus() == GBNSC_SWITCH_DEAD && pingReceived) {
         // perform the initialization protocol
         // notify the parent and Jammer that the switch has been reset
   } else
   if (GBNSC_State.getStatus() == GBNSC_SWITCH_RUNNING && missedPings >= 3) {
      GBNSC_GlobalState.setSwitchStatus(GBNSC_SWITCHSTATUS_DEAD);
      // notify the parent and Jammer that the switch is down
   }

   pingReceived = resetOccurred = FALSE;
   cell.readMR(...);
   cellInterface->sendCell(cell);
   scheduleTimeout(10);
}
```

Figure 9. GBNSC_PingProtocol Definition.

This causes the CMDATA field to be set to this value (as explained in Section 5.1, certain CMDATA values are predefined constants used for GBNSC internal functions, with all the other values being available for use by NCCP-initiated operations). The constructor then initializes the other data fields. The destructor does not have to do anything.

The definitions of the two required virtual functions are also shown in Figure 9. The *receiveCell()* function is called when a cell is received. The function sets pingReceived to TRUE, resets the count of missed pings, and sets resetOccurred to the value contained in the INFO field returned from the switch (obtained through the access functions of the cell class).

Opcode	Command	Description
0	NOP	No operation (used for cells that test switch operation and internal paths)
F0 (hex)	RST	Hard reset of all chips. The opcode is F0 instead of 1 so that a single bit error in a NOP control cell does not transform it into a RST.
2	CLRERR	Clear all error flags in all chips
3	RDVPXT	Read virtual path table entry from VXT (everything except CC field)
4	RDVCXT	Read virtual circuit table entry from VXT (everything except CC field)
5	RDVPXTCC	Read cell counter (CC) from virtual path table
6	RDVCXTCC	Read cell counter (CC) from virtual circuit table
7	WRVPXT	Write virtual path table entry into VXT (does not write CC field)
8	WRVCXT	Write virtual circuit table entry into VXT (does not write CC field)
9	WRVPXTTR	Write virtual path table entry into VXT and start transitional time stamping
10	WRVCXTTR	Write virtual circuit table entry into VXT and start transitional time stamping
11	WRVPXTCC	Write cell counter (CC) to virtual path table (for testing only)
12	WRVCXTCC	Write cell counter (CC) to virtual circuit table (for testing only)
13	ERRORS	Return a cell only if error conditions exist
14	RDMR	Read maintenance register field
15	WRMR	Write maintenance register field

Table 1: WUGS Control Cell Opcodes

The *timed_out()* function is the heart of the operation. When it is called, it first determines if the most recent ping cell was missed. If so, it increments the count of missed pings. It then determines if the switch must be re-initialized, meaning that either a reset occurred or the switch was dead but is now back up. If so, it performs the initialization protocol and notifies the CMSS parent and Jammer that the switch has been reset. Otherwise it checks to see if the switch is dead, meaning that three pings have been missed. If so, it sets the status appropriately and notifies the other processes. After these checks, it prepares the variables for the next ping operation, formats a cell to read the desired maintenance register, sends the cell, and re-schedules itself to time-out in ten seconds.

4.2.2. The GBNSC_RequesteeBase Class

As shown in Figure 8, the GBNSC_RequesteeBase class is derived from the GBNSC_OperationBase class and the NCCP_RequesteeBase class. One class is derived from the GBNSC_RequesteeBase class for each type of operation that may be initiated by another process (the CMSS parent or Jammer).

The derived-class constructor is required to take (at least) a channel address and a ByteBuffer as arguments, and invoke the GBNSC_RequesteeBase constructor with the channel address. The address is in turn passed to the NCCP_RequesteeBase constructor, so the source of the request can be set. The derived constructor should then use the retrieve function to read the request message from the ByteBuffer. Virtual *retrieve()* and *handle()* functions must be provided for the NCCP derivation, and virtual *receiveCell()* and *timed_out()* functions for the operation derivation.

A virtual *initiate()* function may also be provided. Its use is described below.

The class adds support for keeping track of the operations. Specifically, the GBNSC maintains three doubly-linked lists of operations initiated by another process. The first list contains operations that must be initiated, the second contains operations in progress, and the third contains completed operations. The function *toBeInitiated()* puts a requestee object on the first list, *inProgress()* puts it on the second, and *completed()* puts it on the third.

When a request message comes in, a requestee/operation object (derived from the GBNSC_RequesteeBase class) is created and the object's *handle()* function (from NCCP_RequesteeBase) is called. Depending on the operation, three things may then occur. If the operation can be quickly performed (*e.g.*, a local ping to test if GBNSC is alive), the handle function can perform it immediately and return TRUE to indicate the object should be deleted. If the operation cannot be performed immediately but can be initiated simply (*e.g.*, a switch ping that involves sending one cell to the switch), the handle function can initiate the operation, store the operation on the in-progress list, and return FALSE to indicate the object should not be deleted. If the operation is rather complex to initiate, the *handle()* function can store the operation on the to-be-initiated list and return FALSE to indicate that the object should not be deleted. These three protocols are recommendations only, and the *handle()* function can use any appropriate method — for example, all operations could be managed according to the third protocol. The important thing is that operations be placed on the appropriate list.

The GBNSC, as part of its main loop, examines the to-be-initiated and completed lists. Any operations on the to-be-initiated list are removed and their virtual initiate function (a member of the GBNSC_RequesteeBase class) is called. The latter function should start the operation by sending necessary control cells to the switch, then place the object on either the in-progress or the completed list as appropriate. Operations on the completed list are deleted.

When cells arrive for the operation, the receiveCell function (from the GBNSC_OperationBase class) is called. If receiving the cell completes the operation, this function should send a response to the requesting process and move the operation to the completed list. Otherwise the function should perform whatever processing is necessary (*e.g.*, store values from the cell and reset the time-out) and leave the cell on the in-progress list.

A typical sequence of events in the processing of an operation initiated by Jammer or the CMSS parent might be as follows:

1. In NCCP_Global.touch() the request message is received, a requestee/operation object of the appropriate type is created, and the object's handle function is called.

2. The object's handle function places the object on the to-be-initiated list and returns FALSE.

3. In the GBNSC main loop, the object is removed from the to-be-initiated list and its initiate function is called.

4. The object's initiate function sends cells to the switch, sets up a time-out, and places itself on the in-progress list.

5. The cell manager receives a control cell from the switch, looks up the CMDATA value, and calls the object's receiveCell function.

6. The object's receiveCell function creates and sends a response to the requester, then places the object on the completed list.

7. In the GBNSC main loop, the object is removed from the completed list and deleted.

Note that the in-progress list is not examined by the GBNSC control logic, and in particular there is no mechanism whereby "old" operations are detected and deleted by the GBNSC. All derived classes must therefore ensure that they eventually arrange for their deletion by placing the object on the completed list.

5. GBNSC_CellManagerClass and the Cell Manager

The GBNSC_CellManagerClass encapsulates the switch's interface to the switch controller. This interface uses an ATMCard object, which uses a NATM socket to communicate with the WUGS. The switch controller contains one static object of this class, called theCellManager.

The cell manager provides access functions whereby the switch controller can reserve or release the send and receive control VXIs. The first of these functions creates the ATMCard interface; the others use the ATMCard protocol to reserve (or free) the VXIs. Once the switch controller has reserved the send VXI, it is the only process on the machine that can use that VXI; and once the receive VXI is reserved, any cells arriving at the machine on that VXI will be delivered to the switch controller.

The cell manager also provides a function to send a control cell. The cell manager passes the 48 byte cell buffer to the atmCard object for delivery to the switch via the native ATM socket.

Finally, the cell manager provides a touch function which receives and processes a cell (if one is available). The cell VXI is compared to the control receive VXI; if they are identical, the cell is a control cell. Its CMDATA field is extracted and provided to the CMDATA manager (Section 5.1), which returns a pointer to the operation that is assigned that CMDATA value. The *receiveCell()* function of that operation is then called with the cell.

The cell manager also supports the sending and receiving of data cells. This feature is useful in generating data traffic in the switch for testing purposes. The mechanism is very similar to that used for control cells. Operations request receive and transmit VXIs as needed and then can send and receive data cells to perform their functions.

5.1. GBNSC_CMDataManager Class

The GBNSC_CMDataManager class provides a reasonably safe way of performing a rendezvous between ATM control cells returning from the switch and the operation objects which should receive them. Unfortunately the extremely safe mechanism provided by the common-code rendezvous object cannot be used with control cells, since the rendezvous information in the cell — the CMDATA field — is only four bytes long, insufficient to hold a rendezvous marker object. The mechanism provided by the CMDATA manager, while not completely secure, is safe against single-bit errors in the CMDATA field and reasonable delays in cell arrival times.

The class provides three functions. The *obtain()* function takes a key value (described below) and a pointer to a GBNSC_OperationBase object, generates a CMDATA value that is associated with that pointer, and returns the CMDATA value. The *release()* function takes a CMDATA value and removes any associated pointer value. The *lookup()* function takes a CMDATA value and returns the associated pointer; a NULL pointer is returned if no pointer is associated with the value. The GBNSC has exactly one instance of the class, which is globally available.

5.1.1. CMDATA Formats

CMDATA values are divided into two groups. One group is for the GBNSC's internal functions (pings, statistics gathering, and so forth), while the second group is for operations initiated by a request from another process. Figure 10 shows how the 32-bit CMDATA field is interpreted for these two groups. (There is nothing magical about these exact formats, and the field sizes could easily be changed if it becomes necessary.)

The high-order bit of both formats, labelled P, is a parity bit computed as the negated XOR of the other 31 bits of the field. This is used for detection of single-bit errors in incoming control cells. If a control cell arrives with a corrupted CMDATA field, it is silently discarded. The low-order eight bits of both formats, labelled index, are used to distinguish between the two.

The low-order eight bits of the CMDATA values corresponding to GBNSC internal operations are set to hexadecimal 0xFF (all 1's). The remaining bits are divided into the operation field and the port field. The seven-bit operation field indicates what type of internal operation is being performed, e.g., ping, report errors, read of IPP statistics field A, and so forth. The sixteen-bit port field is used with some of the operations (such as statistics reads) to indicate

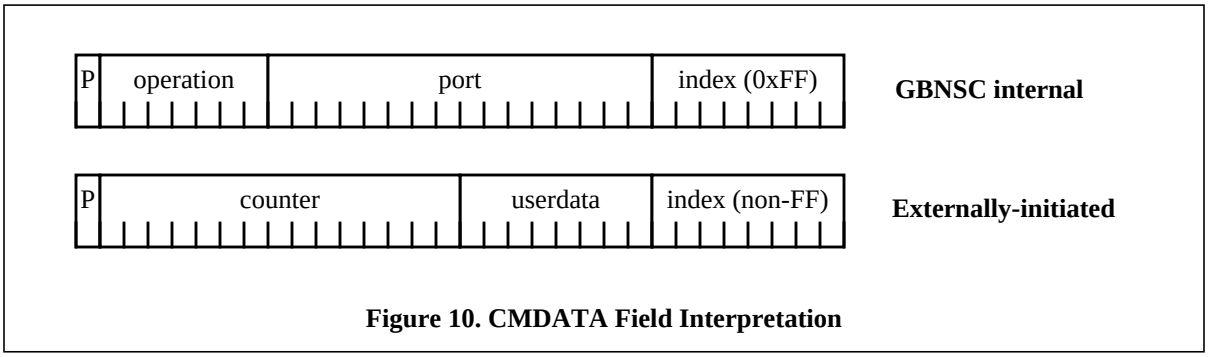


Figure 10. CMDATA Field Interpretation

which port was affected. This allows a single object to manage an operation for the entire switch. For example, the object which handles reading IPP statistics field A can send out eight control cells (in an eight-port switch), placing the appropriate port number into each. As each control cell returns, the object can examine the port number to determine which port was read by the cell and update the corresponding GBNSC statistics table entry.

The low-order eight bits of the CMDATA values corresponding to externally-initiated operations may have any value other than 0xFF. This eight-bit value is an index into a table inside the CMDATA manager; its use is described below. The remainder of the cell is divided into a 15-bit counter whose use is also explained below, and an 8-bit user data field. The latter field may, like the port field of the GBNSC internal format, be used by an operation to mark cells for its own use.

5.1.2. CMDATA Allocation, Cell Rendezvous, and Deallocation

When an operation object (derived from the GBNSC_OperationBase type) is created, it is supplied a key which may be GBNSC_CMDATA_NONE, GBNSC_CMDATA_EXTERNAL, or one of the predefined operation types such as GBNSC_CMDATA_PING. The key GBNSC_CMDATA_NONE indicates that the object will not require a CMDATA value, so the object does not interact with the CMDATA manager. Other keys are used to request a CMDATA value from the manager; a pointer to the object is supplied as well as the key.

If the key is one of the predefined operation types, the manager records the object pointer in an internal variable and returns a CMDATA value in which the index is 0xFF, the operation field is the indicated value, the port is 0, and the parity field is the NXOR of the other bits. There is one such internal variable for each of the predefined operation types, meaning that there can be at most one object of that type active at any time. Since, as explained in Section 4.2.1.1, most of these operations are managed by a permanent object that schedules itself for regular activation, this is an acceptable design.

If the key is GBNSC_CMDATA_EXTERNAL, the manager examines a structure of 255 records. Each of these records contains an operation pointer and a counter. The manager examines the table and locates an entry with a NULL operation pointer. If no such entry is found, an error occurs. If a NULL pointer is found, the manager stores the operation object's pointer in the table and constructs a CMDATA value in which the index field is the table index, the counter field is the counter stored in the table, the userdata field is 0, and the parity field is the NXOR of the other bits. This value is returned to the operation. The table entry remains in use until the operation is destroyed. Note that this means there can be no more than 255 simultaneously-active operations initiated by other processes; if this is found to be too small, the index field can be widened at the expense of the counter field.

When a control cell arrives from the switch, the cell interface manager asks the CMDATA manager for the operation associated with the cell's CMDATA value. The CMDATA manager first checks that the parity field is the negated XOR of the other bits. If the check fails, the manager returns a NULL pointer. If the parity check succeeds, the CMDATA manager next checks the index field. If the value is 0xFF, it examines the operation field and returns the pointer associated with the value in that field. If the operation value is illegal, a NULL pointer is returned. If the index field is not 0xFF, the CMDATA manager compares the counter field with the counter field in its internal table at the index. If the two are equal, it returns the pointer stored in that table entry; if they are unequal, it returns a NULL pointer.

If the CMDATA manager returns a non-NULL operation object pointer, the cell interface manager uses the pointer to call the object's *receiveCell()* function. If a NULL pointer is returned, the cell interface manager silently discards the cell. The operation to which the corrupted cell should have been directed will presumably time-out and either retry the operation or report a failure.

When an operation object is destroyed, it releases its CMDATA value (if it has one). The CMDATA manager examines the index portion of the value and if it is 0xFF, it uses the operation field to set the appropriate internal pointer to NULL. If it is any other value, the manager sets the pointer in its internal table at that index to NULL and increments the counter modulo 2^{15}. Thus, the next operation that is assigned that particular index will get a different counter value, so any cell from the previous operation that is (extremely) delayed in returning to the switch will not be directed to the new operation.

5.1.3. Implementation for Safer Rendezvous

This section discusses some ways of reducing the probability of a serious error in cell rendezvous, defined as the cell being sent to and processed by the wrong operation object. Whether these ideas are worth adding to the implementation depends on the perceived danger of this happening. Perhaps the best way to decide whether to add features is to see how often the problem arises, and add some or all of them if the error rate is unacceptably high.

The operation's *receiveCell()* function is one place where the error can be detected. The function should normally have some idea what type of cell it will be receiving (*e.g.*, WRVCXT on a particular VCI) and can check the cell it receives to see if it is that type. If not, the operation can discard it and wait for the correct cell.

One way that an invalid rendezvous could occur is for an operation to send out a cell, be destroyed, and a new operation be assigned the same CMDATA value before the cell returns. The cell would then be incorrectly delivered to the new operation. This is extremely unlikely for externally-initiated operations, since it means that the 15-bit counter wrapped around completely. Since the 255 index slots are (approximately) used in a round-robin fashion, a total of $255*2^{15}$ operations would have to have occurred between sending and receiving the cell. If we assume the CP is processing 1000 operations per second, this would be a delay of somewhat over two hours. The probability of this can be ignored (unless, of course, the CP is on Earth and the switch is orbiting Saturn).

The probability of a reused CMDATA value is rather higher for the GBNSC-initiated operations, since they are always assigned the same CMDATA values. However, these operations are for the most part permanent — once created, they remain around forever, and no other operation of that type is ever created. Even if, for some reason, the operation were destroyed and another one created, in most cases it would not matter if the new operation received a cell from the old one —a RDMR STATISTICS-A cell for IPP 6 is useful to the IPP statistics-A gathering operation no matter where it came from.

Some slight adjustments to the CMDATA manager implementation may reduce the probability of a bad rendezvous due to corruption of the CMDATA field. The parity field protects against single-bit errors, so the adjustments should be such that multiple-bit errors are less likely to produce a valid CMDATA value.

There is not too much that can be done with the GBNSC internal operations, since half of the field is the port number which is not checked by the CMDATA manager. Fortunately these are not a serious concern, since each of these operations will be generating only one type of cell — NOP, RDMR STATISTICS-A, etc. — and the *receiveCell()* function can easily check this. Some security can be obtained by spreading out the constants within the seven-bit field, perhaps selecting them so that they are all at least three bit errors apart.

Several simple changes can reduce the probability of a corrupt CMDATA value producing a valid number for the externally-initiated cells. There are several ways that this might occur:

- The 0xFF (index) portion of a GBNSC internal CMDATA is corrupted. It is highly unlikely that this will produce exactly the counter that is stored at that index. However, for the truly paranoid, the routines that adjust the counter could be set up so that the upper seven bits of the counter (which correspond to the operation type in the other format) are never equal to the constants used for the operation types.

- The index portion of an externally-initiated CMDATA is changed to another index, and the counters happen to match. The probability of this occurring can be reduced by initializing the counters to values that are reasonably far apart — say, starting each counter at index*50. Since the slots are (approximately) used equally-often, the counters will remain reasonably far apart.

- An externally-initiated operation sends a cell and then is deleted, another operation is assigned that index, and the cell arrives back with the counter portion corrupted so that it happens to be the one assigned to the new operation. This could easily happen — for example, the count 1 could be changed to 2 by a two-bit error which would not be detected by the parity check. Changing the counter increment might help in this case; for example by incrementing the count by 13, all 2^{15} counter values will still be used but at least three bit errors are needed to change a counter into the next one allocated.

6. GBNSC Initialization

The GBNSC process is started either directly from the command line or when its CMSS parent creates an NCMO object. The NCMO constructor is supplied with the name of a configuration file which describes either a node or a switch. The NCMO object creates a socket on which it will accept an incoming connection from the GBNSC and then starts the GBNSC process, supplying it its identification information, the configuration file name and the TCP socket port.

[OLD VERSION: The NCMO object creates a PriquePair which will be used to communicate with the child process and starts the GBNSC process, supplying it the configuration file name, PriquePair identifiers, and simulation slot number via command-line arguments.]

The GBNSC initialization has three main stages, described in detail in the following sections. In the first stage, the GBNSC reads the configuration file and initializes its communications systems. In the second stage, it obtains a control connection for its switch, initializes the hardware and starts the internal operations (such as the ping protocols). Finally, the process informs its CMSS parent that it is in control of the switch and goes into its input-driven main loop.

6.1. Configuration File and Communications Initialization

(CONFIG FILE description probably needs to be checked against actual current configuration file.)

As described in Section 3, the GBNSC uses two protocols (NCCP and ATMCard) to communicate over two TCPObject channels. The GBNSC creates the TCPObject channel to the NCMO parent first, before attempting to read the configuration file. If this connection fails, the GBNSC process exits and the CMSS parent receives a child-died signal from the operating system. If the connection succeeds, the GBNSC process registers the TCPObject channel with the NCCP global object so that the latter object will check for NCCP messages on the channel.

After establishing the communications channel to the NCMO parent, the GBNSC reads and parses the configuration file. Figure 11 shows a sample configuration file for the GBN Switch Controller.

The first section of the configuration file describes the switch hardware features, that include number of ports, IPP and OPP chip types for each port. The TCPPORT entry indicates what port the switch controller should attempt to listen on for new incoming connections. The next two entries are the IPP and OPP default chip types. The GBNSC stores these default values in each slot of the internal tables. The default chip types may be overridden for specific chips, as shown in Figure 11; the keyword END must be present, even if there are no overrides. After any overrides are read and stored in the tables, the GBNSC accesses hardware information files to read information about the chips such as VXT table size and default values of maintenance register fields. It then creates initial values for the processor configuration maintenance registers.

The next portion of the file is a list of global parameters for the switch controller. These parameters appear within the keywords PARAMS and END. The sample file in the figure defines the VPT value as 1 (the switch will be set up so all VPI/VCI pairs of the form 1/X are interpreted as VC connections, and all others as VP connections) and the

```
SWITCH 1 GBN            --
{
        -- Switch hardware section

        TCPPORT 3550            -- use this port for TCP communications
        PORTS 8                 -- 8 ports, numbers 0 to 7
        CHIPS 1 2               -- Default chips, IPP and OPP
              IPP CHIP 3      -- Overrides for specific ports: chip number,
                       6       -- then ports
         END

        PARAMS
            VPT 1                           -- This VPI is VP-terminated
            CONFIGURATION TIMEOUT 60        -- 60 seconds to configure
            POLLING TIMEOUT 60              -- 60 seconds between polling
        END

        CONTROL
            0 0             -- CP is connected to these two ports
            0/32            -- control VXI (cells from CP)
           0/32            -- control return VXI (cells arriving at switch)
        END

        LINKS
        -- Port Direction Type Speed   Resys-bw       Client
            0     <=>     UNI  @155   @2200         "r2d2.arl.wustl.edu"   1
            1     <=>     UNI  @155   @2200          "owen.arl.wustl.edu"  1
            2     <=>     NNI  @620   @1780    0 5  “jabba.arl.wustl.edu”
            3     <=>     NNI  @620   @1780
            4     <=>     NNI  @620   @1780
            5     <=>     NNI  @620   @1780
            6     <=>     NNI  @620   @1780
            7     <=>     NNI @2400   @0
        END
        INIT
           IPP 0 VPCOUNT 255          -- Use this value for VPCount in IPP 0
            IPP 1 VPCOUNT 127
            IPP 2 VPCOUNT 255
            IPP 3 VPCOUNT 255
            IPP 4 VPCOUNT 255
            IPP 5 VPCOUNT 255
            IPP 6 VPCOUNT 255
            IPP 7 VPCOUNT 255
        END

        PRESET
        --              Input                   Output
        --      Port    VPI     VCI     Port    VPI     VCI
                 0       1       34      4       0       32
                4       0       0       32      1       34
        END
}
```

Figure 11. Sample GBNSC Configuration File.

configuration time-out duration as 60 seconds (the controller will allow this much time to get the switch ready for use). All parameters have a default value, so this section (including PARAMS and END) may be completely omitted.

The next portion of the file describes the control connections that are to be used by the GBNSC. These connections are determined in advance by the network managers, who must guarantee that they can be realized. For the file shown in the figure, the GBNSC is told that it will be sending cells to IPP 0 and receiving them from OPP 0. It should put the VXI 0/32 in control cells that it sends to the switch IPP 0. The GBNSC will receive returning control cells with the VXI 0/32 at OPP 0 of the switch.

The next section of the file shown in Figure 11 describes the local topology of the switch, that is, the entity connected to each of its ports. Each entry begins with a switch port number, followed by a symbol indicating whether the connection is from the port to the other entity, from the other entity to the port, or bidirectional. The other end of a link may be either a terminal or another switch. Terminals are identified by the keyword UNI followed by the link bandwidth and a string identifying the terminal (in the example, machine names are used). A link to another switch is shown by the keyword NNI followed by the link capacity information. There are two types of NNI record format. The long format gives the other end information including the other side switch id, the other side link id, and the other side CP name. The short format does not contain other side information. The other side information will be retrieved by a link level protocol, which we name Hello Protocol (to be documented and implemented later). Link capacity is given by two integer numbers, in Kbits/s, indicating the external capacity and recycling capacity.

The next section of the configuration file describes the initial configuration that the switch is to use. The values here override the default values calculated from the switch size, chips, and links. In the figure, the managers have decided that the VPCOUNT register of IPP 1 is to contain the value 127 (rather than the default value, 255). This section may be absent.

The final section of the configuration file describes a list of preset connections that the SC uses to communicate with its neighbor switches. These connections can be used as signaling channels, Hello protocol channels or control connections for subsidiary switches. In the example, we set a channel (port 0 VXI1/34 to port 4 VXI0/32) to control the next switch. (This feature is not yet implemented.)

6.2. Setup of Control Connections and Switch Initialization

In the next phase of the initialization, the GBNSC must establish the control connections to its switch. From the configuration file, the GBNSC knows the switch ports and VXIs to be used. The GBNSC process is also able to determine on which terminal (machine) it is running. The problem thus becomes one of setting up a connection from that terminal on the send VXI to the switch input control port on the control VXI, and a reverse connection from the switch output control port on the RHDR VXI to the terminal on the receive VXI. There are two cases:

1. The GBNSC's machine is directly connected to the switch control ports, and the send and control VXIs are the same as are the RHDR and receive VXIs. The GBNSC is in direct control of the switch, and simply has to reserve the send and receive VXIs with the ATMCard object. It should also internally reserve the RHDR VXI for the output control port so that no other process can use it. (This is the default in the current implementation as the VXIs are allocated as bidirectional.)

2. The switch control ports are connected to other switches in the same node. In this case, the switch which is closer to the CP should come up first. The control channels that control subsidiary switches have to be set up first. In the sample configuration file in Figure 11, switch 2 is connected to port 4 of switch 1. When switch controller 1 comes up, it sets channel VXI1/34 as the control channel for switch 2. Switch 1 reserves the send and receive VXIs with the ATMCard object and internally reserves the RHDR VXI for the output control port. (This process must be done manually in the current implementation.)

Once the control connection is established, the GBNSC initializes the switch. It first pings the switch (sends a NOP cell through it) to make sure the switch is alive and to roughly determine the turnaround time. It then sends a series of WRMR cells for the configuration and hardware fields, writing the initialization information into the switch maintenance registers. The protocol then initializes the VXT entries for the terminated VPI that is to be used for VC connections, setting the VPT bit in each. Finally, the protocol creates the permanent operation objects for the GBNSC internal operations, such as the PingProtocol object of Figure 9.

6.3. Initialization Wrap-Up

During the initialization, several things can go wrong — the setup of the communications channels might fail, the configuration file might be syntactically malformed, the hardware files described in the configuration file might be missing or corrupted, the control connections might be illegal, and so forth. If any of these occurs, the GBNSC process terminates itself. If it has established the communications channel to its parent, it creates an NCCP ChildStatus requester object with an error status and calls the object's *send()* function. This sends a message to the parent process informing it that the GBNSC has died and the reason for the failure.

If no failure occurs in the initialization phase, the GBNSC will send a series of NCCP_LinkStatus message [29] to its CMSS parent to report link status and the other end information. The other end information may either come dynamically from the Hello protocol (will be documented and implemented later) or come statically from the configuration file. Then GBNSC creates an NCCP ChildStatus [29] object with a ready status and sends it to its CMSS parent, indicating that the switch is ready for use. It then enters the GBNSC main loop.

7. Connection Related Operations

For a switch controller, there are three types of connection related operations that are requested by the CMSS parent [29]: ReserveMP; UpdateHardwareMP; and RollbackMP. A ReserveMp operation requests the SC to reserve required resources for a connection. An UpdateHardwareMP operation commits the reserved resources into the switch hardware. A RollbackMP operation brings the object state back to previously committed state. In this section, we first introduce the objects that support these operations. Then we give some examples to show how multipoint-to-multipoint communications are supported.

7.1. Connection State Objects

7.1.1. GBNSC_Link

Information for a switch link is distributed in three objects: GBNSC_Link, GBNSC_IPP and GBNSC_OPP.

A GBNSC_Link object holds information for an external link of the switch. Figure 12 shows the attributes of a

```
un4byte             linkPort;          // prot ID
SC_LinkType         linkType;          // UNI or NNI
SC_LinkDirection    linkDirection;     // IN, OUT, or Bidirectional
un4byte             linkRate;          // K bits/s
un4byte             linkRecycRate;     // K bits/s
char*               termName;          // Indicate the machine on the other side
un4byte             otherNodeId;       // In case of NNI, it gives the other end switch
un2byte             otherLinkId;       // In case of NNI, it gives the other end link id
```

Figure 12. GBNSC_Link Attributes

link object. Most of the information comes from the configuration file. Some, such as the other end information, may come from the Hello Protocol. The GBNSC_SwitchController object maintains an array of GBNSC_Link objects indexed by link Id. Similarly, the GBNSC_SwitchController maintains an array of GBNSC_IPP objects and an array of GBNSC_OPP objects.

7.1.2. GBNSC_IPP

A GBNSC_IPP object holds the information for an input port. A GBNSC_IPP object contains a VXIManager and a BandwidthManager. The VXIManager maintains the reserved and free VXI entries. By calling the VXIManager's reserve and free functions (*reserveVP(), reserveVC(), freeVC()* and *freeVP()*), VXI entries can be reserved or

released. The BandwidthManager keeps track of bandwidth usage on the input link. The BandwidthManager could be replaced by a QOSManager which will do more complex resource allocation. Current GBNSC_IPP attributes and VXIManager and BandwidthManager's access functions are listed in Figure 13.

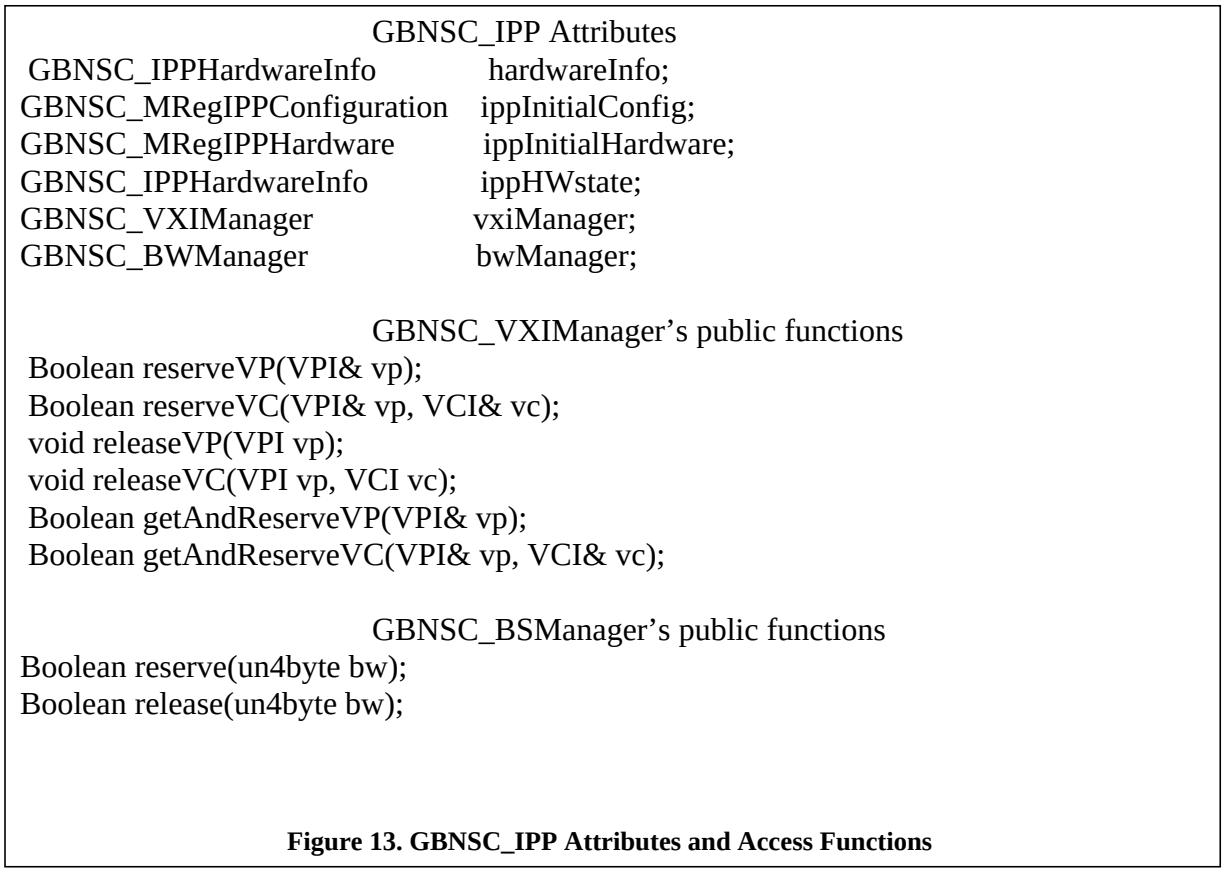


Figure 13. GBNSC_IPP Attributes and Access Functions

The hardwareInfo, ippInitialConfig, ippInitialHardware and ippHWstate fields store the hardware information for the IPP chip. The GBNSC_SwitchController periodically reads the hardware state of each chip on the switch and sets these fields.

7.1.3. GBNSC_OPP

A GBNSC_OPP object holds the information for an output port. A GBNSC_OPP object contains a VXIManager and a BandwidthManager as in GBNSC_IPP. Current GBNSC_OPP attributes are listed in Figure 13.

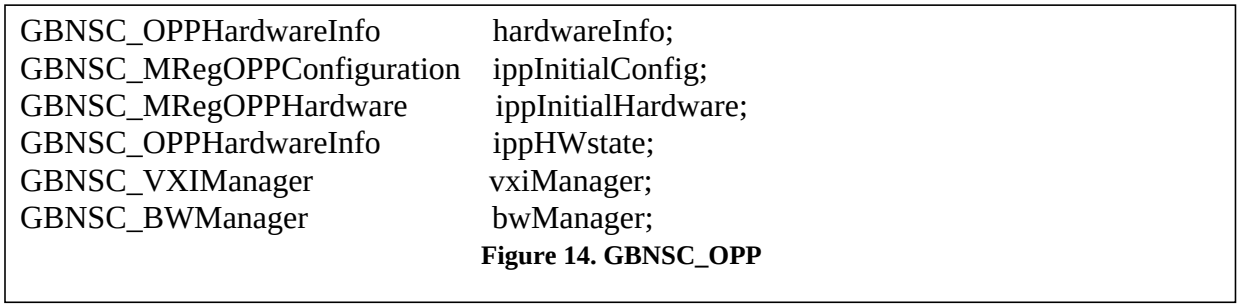


Figure 14. GBNSC_OPP

7.1.4. GBNSC_Multipoint

A GBNSC_Multipoint object represents a multipoint-to-multipoint connection in a WUGS. A multipoint connection is implemented in a WUGS as a recycling tree [27]. Depending on the number of sources (the entrance of data) and sinks (the exit of data), the recycling tree may take different forms as shown in Figure 15.

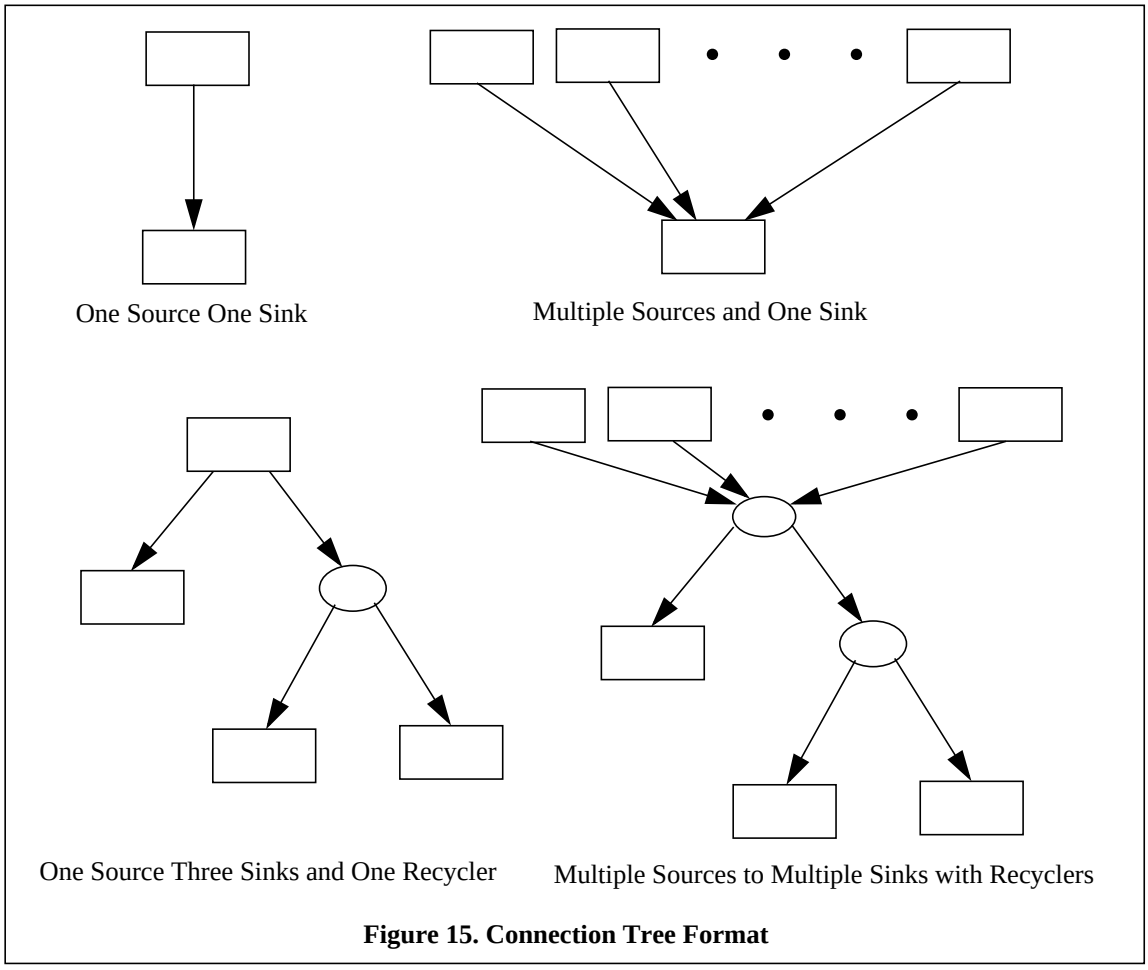


Figure 15. Connection Tree Format

When there is only one sink, data from any source can be directly transferred to that sink. When there are multiple sources and multiple sinks, we have to use N-1 recycling ports where N is the number of sinks. To represent the recycling tree structure in the switch, a GBNSC_Multipoint maintains a GBNSC_ConnectionTree object. The GBNSC_ConnectionTree object contains a list of source objects and a binary tree whose elements are sinks as leaves and recyclers as internal nodes. To make it easier to connect sources, sinks and recyclers to form a connection tree, GBNSC_Source, GBNSC_Sink, and GBNSC_Recycler are all derived from a base class, a GBNSC_TreeElement class, which in turn is derived from NCMO_SourceSinkBase which defines some common access functions and attributes. We discuss the details of GBNSC_Source, GBNSC_Sink, and GBNSC_Recycler in following sections.

Besides the connection tree, a GBNSC_Multipoint contains attributes and access functions as shown in Figure

16. Since multipoint operations may be asynchronous, the requestee field is a pointer that points to the current

```
GBNSC_ConnectionTree     connectionTree;
GBNSC_RequesteeBase*     requestee;

Boolean reserve();
Boolean updateHardware();
Boolean rollback();
void    reportState(GBNSC_TreeElement *, Boolean);
```

Figure 16. Part of GBNSC_Multipoint's Attributes and Access Functions

requestee. When a source, a sink or recycler completes its operations, it calls the GBNSC_Multipoint's *reportState()*. When the last expected response is received, the multipoint object finds the requestee through its requestee pointer and sends a response to its CMSS parent. An operation requestee object (GBNSC_ReserveMP, GBNSC_UpdateHardwareMP, or GBNSC_RollbackMP) could call the GBNSC_Multipoint's *reserve()*, *updateHardware()*, or *rollback()* to start the corresponding operation.

7.1.5. GBNSC_TreeElement and GBNSC_ConnectionTree

GBNSC_TreeElement is a pure virtual class to provide a means to build the connection tree. It basically provides left, right and parent pointers to point to other GBNSC_TreeElements. Associated with the left and right pointer is a count that counts the number of descendents along the pointer. When a new sink is to be added, the tree is traversed along the path with smaller descendent count, thus the connection tree will only increase its depth only if necessary. This helps reduce the delay in the switch.

A GBNSC_ConnectionTree object has a root, initially NULL, a source list. Sources and sinks can be dynamically added or removed through *addSource()*, *addSink()*, *removeSource()*, *removeSink()* operations. The above discussion may give the impression that the connection tree is built while the sinks or sources are created and reserved. That is not true. The reserve operation only verifies that the resources needed for the object(s) are available. Sinks and sources are added into the connection tree only when the resources are to be committed into the hardware. So any call of *addSource()*, *addSink()*, *removeSource()*, *removeSink()* will cause the corresponding control cells to be sent to the switch.

7.1.6. GBNSC_Source, GBNSC_Sink and GBNSC_Recycler

GBNSC_Source, GBNSC_Sink and GBNSC_Recycler are all derived from GBNSC_TreeElement which, in turn, is derived from NCMO_SourceSinkBase. As a derived object, a GBNSC_TreeElement has to implement the following functions.

1. *reserve()* - check and reserve the bandwidth and VXI resources.
2. *unreserve()* - release bandwidth and VXI resources.
3. *commit()* - mark the element state as committed.
4. *rollback()* - undo the previous uncommitted operation.
5. *allocHardware()* - send control cells to implement reserved connection channel.
6. *releaseHardware()* - send control cells to release an established connection channel.

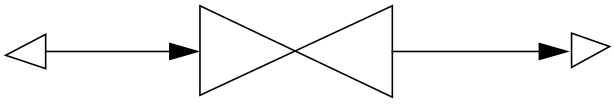
7.2. Connection Oriented Operation Scenarios

In this section, we give some connection-oriented operation scenarios to show how they are implemented in the GBNSC.

7.2.1. A NCCP_reserveMP Received with One Source and One Sink

- An NCCP_reseveMP message is received, NCCP_RequesteeBase::*baseHandle()* is called which generates a GBNSC_RequesteeReserveMP object. The latter object's *handle()* function gets called.
- The *handle()* function first tries to rendezvous with the GBNSC_Multipoint indicated in the NCCP_reserveMP message. In this case, since this is the first message for the multipoint, the rendezvous fails.
- The *handle()* function creates a GBNSC_Multipoint MP and calls its *reserve()* function.
- The MP's *reserve()* function scans through the message list and creates new GBNSC_Source and GBNSC_Sink objects.
- The source's and sink's *reserve()* functions get called.
- The source's and sink's *reserve()* functions have their VXIManagers and BWManagers check and reserve the required resources. If the reservations fail, their *rollback()* functions are called and the failure return of *reserve()* causes the MP's *rollback()* to get called which will roll the entire MP state back to the previously committed state. If the reservation succeeds, the *reserve()* returns TRUE.
- When all the sources and sinks involved in the current operation are successfully reserved, an NCCP_reserveMP response is sent back to the CMSS parent with a rendezvous marker that indicates the MP.
- The GBNSC_RequesteeReserveMP object is deleted.

The result of the operation is shown in Figure 17.



MP, one source, one sink reserved and not committed
No connection tree built for the MP

Figure 17. First NCCP_reserveMP Processed

7.2.2. A NCCP_UpdateHardwareMp Received

- An NCCP_UpdateHardwareMP message is received, NCCP_RequesteeBase::baseHandle() is called which generates a GBNSC_RequesteeUpdateHardwareMP object. The latter object's *handle()* gets called.
- The *handle()* function tries to rendezvous with a GBNSC_Multipoint. Since the MP exists this time, the rendezvous mechanism calls the MP's *REQ_rendezvous()* which in turn calls the MP's *updateHardware()* to process the operation request.
- The *updateHardware()* scans through the commit list of sinks and sources. The *commit()* function of each

source and sink (one for each in this case) is called and it is added into the connection tree by connectionTree.*addSource()* or connectionTree.*addSink()*. In the current example, the sink is the first sink to be added into the connection tree (case 1 in Figure 15). In this case, no control cell is sent in connectionTree.*addSource()*.

- After sources and sinks are added into the connection tree, *updateHardware()* calls *commitSources()* which causes the sending of control cells to the switch. The UpdateHardwareMP operation is asynchronous in the sense that it does not respond until all control cells have returned.
- In the GBNSC main loop, GBNSC_CellManager::*touch()* is called to check for incoming cells. When an incoming cell is received, it picks the CMData from the control cell and rendezvous with the pending object, a source or a sink or a recycler.
- The pending object's *receiveCell()* is called. If this is the last control cell the object is waiting for, it calls the MP's *reportState()* to report the status of the processing.
- When the MP finds that all the pending operations are done, the response is sent to the CMSS parent and the current requestee is deleted.

The operation result is shown in Figure 18.

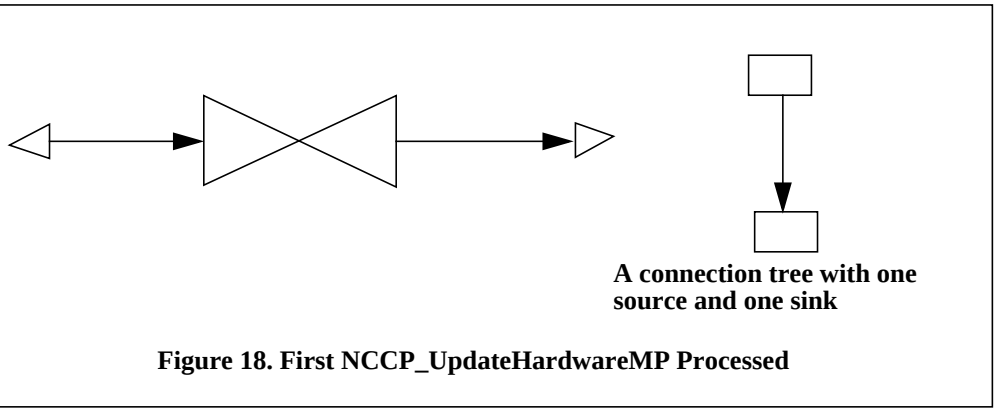


Figure 18. First NCCP_UpdateHardwareMP Processed

7.2.3. A NCCP_ReserveMP Received with One More Source

When an NCCP_ReserveMP is received with one new source, the source is added into the MP's source list and the resources are reserved. No actions are performed on the connection tree. If a NCCP_Rollback is received thereafter, the source will be removed from the MP, no control cell will be involved to complete either reserve operation or rollback operation. The operation result is shown in Figure 19.

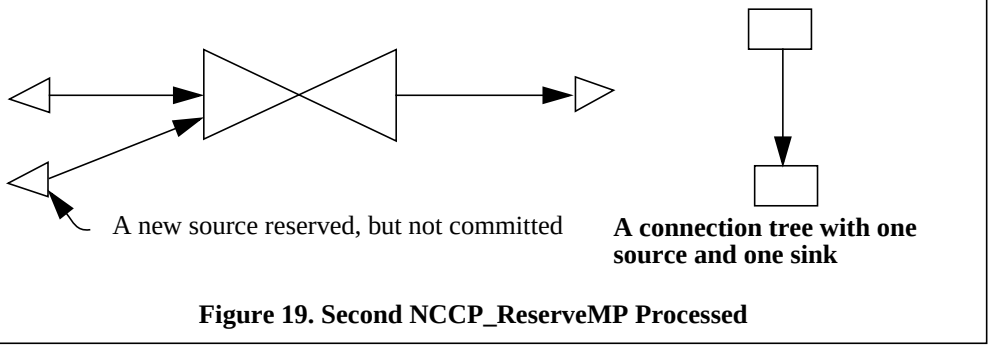


Figure 19. Second NCCP_ReserveMP Processed

7.2.4. A NCCP_UpdateHardwareMp Received

When the new source is to be committed, the connection tree is updated. The processing sequence is quite similar as described in Section 7.2.2. The operation result is shown in Figure 20.

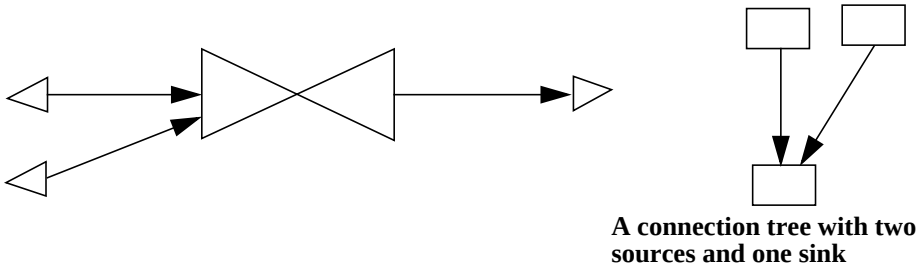


Figure 20. Second NCCP_UpdateHardwareMP Processed

7.2.5. A NCCP_ReserveMP Received with One More Sink

When a NCCP_ReserveMP is received with one new sink, the resources needed are reserved. No action is performed on the connection tree. The operation result is shown in Figure 21.

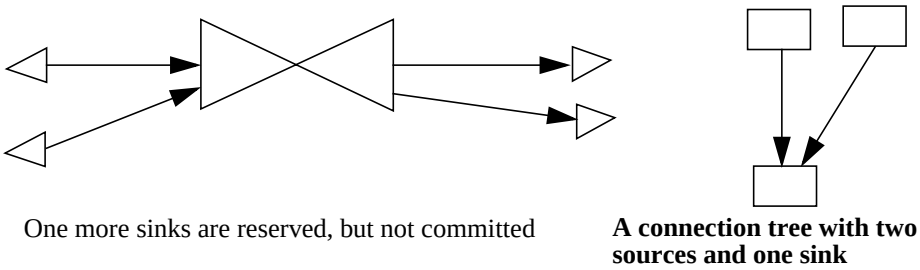


Figure 21. Third NCCP_ReserveMP Processed

7.2.6. A NCCP_UpdateHardwareMp Received

When the new sink is to be committed, the connection tree is updated. The processing sequence is quite similar

to that described in Section 6.2.2. The operation result is shown in Figure 22.

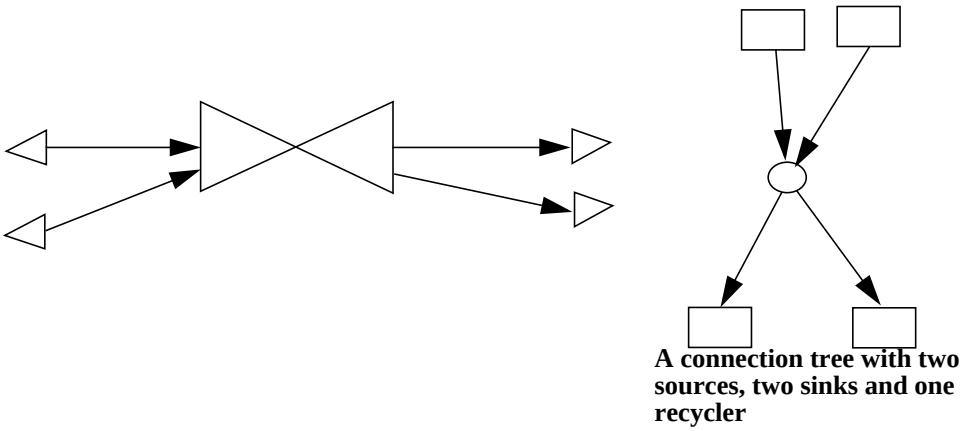


Figure 22. Third NCCP_UpdateHardwareMP Processed

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] ATM Forum, “ATM Forum 94-0471R7 PNNI Draft Specification”, The ATM Forum 1994.

[4] 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.

[5] R. G. Bubenik, J. D. DeHart and M. E. Gaddis. “Multipoint Connection Management in High Speed Networks.” In IEEE Infocom ‘91: Proceedings of the Tenth Annual Joint Conference of the IEEE Computer and Communications Societies, pages 59-68, April 1991.

[6] R. G. Bubenik, M. E. Gaddis and J. D. DeHart. “A Strategy for Layering IP over ATM”. Washington University Applied Research Laboratory, Working Note 91-01, Version 1.1, April 1991.

[7] R.G. Bubenik, M.E. Gaddis, and J.D. DeHart. “Virtual Paths and Virtual Channels.” In IEEE Infocom ‘92: Proceedings of the Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, May 1992.

[8] 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.

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

[10] J. R. Cox and J. S. Turner. “Project Zeus Design and Application of Fast Packet Campus Networks”. Washington University, Department of Computer Science Technical Report 91-45, July 1991.

[11] 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.

[12] J. DeHart, “Washington University GigaBit Network Software Installation and Start-up”, Washington University, Applied Research Laboratory Working Note ARL-96-02, DRAFT, March 1996

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

[14] 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.

[15] M. E. Gaddis, R.G. Bubenik, and J.D. DeHart. “Connection Management for a Prototype Fast Packet ATM B-ISDN Network.” In Proceedings of the National Communications Forum, vol. 44, pp. 601-608, October 8-10, 1990.

[16] M. E. Gaddis, R.G. Bubenik, and J.D. DeHart. “A Call Model for Multipoint Communications in Switched Networks.” submitted for publication to ICC ‘92, Chicago, Illinois, June 1992.

[17] S.E. Minzer. “Broadband ISDN and Asynchronous Transfer Mode (ATM).” In IEEE Communications Magazine, 27(9):17-24, September 1989.

[18] S.E. Minzer and D.R. Spears. “New Directions in Signaling for Broadband ISDN.” In IEEE Communications Magazine, 27(2):6-14, February 1989.

[19] G.M. Parulkar, J.S. Turner. Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment. In IEEE Infocom ‘89: Proceedings of the Eighth Annual Joint Conference of the IEEE Computer and Communications Societies, pages 655-667, April 1989.

[20] G. M. Parulkar. “The Next Generation of Internetworking”. ACM SIGCOMM Computer Communications Review. vol. 20, no. 1, New York, NY, pp. 18-43, January, 1990.

[21] A.S. Tanenbaum. Computer Networks. Prentice-Hall, 1981.

[22] J. S. Turner, “Fast Packet Switching System”, U.S. Patent 4 494 230, January 15, 1985.

[23] J.S. Turner. “New Directions in Communications.” In IEEE Communications Magazine, 24(10):8-15, October 1986.

[24] J.S. Turner. “Design of an Integrated Services Packet Network.” In IEEE Transactions on Communications, 4(8):1373-1380, November 1986.

[25] J.S. Turner. “Design of a Broadcast Packet Switching Network.” In IEEE Transactions on Communications, 36(6):734-743, June 1988.

[26] J. S. Turner. “A Proposed Management and Congestion Control Scheme for Multicast ATM Networks.” Washington University, Computer and Communication Research Center Technical Report 91-01, May 1991.

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

[28] 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.

[29] D. Wu and J. DeHart, “Node Controller Managed Object (NCMO) and Node Controller COmmunication Protocol (NCCP),” Washington University, Applied Research Laboratory Working Note ARL-96-03, June 1996.