Modeling ATM networks: a case study

Włodek Dobosiewicz* Paweł Gburzyński†

(Extended version)

Abstract

We discuss the simulation model of a network consisting of a number of ATM switches. The model illustrates some issues related to simulating ATM networks at the cell level. Owing to its object-oriented design, our model is extensible, i.e., the functional components of switches can be easily redefined to reflect different physical implementations of the ATM concept.

Keywords: ATM networks, simulation, protocol specification

1 Introduction

The problem of investigating the performance of ATM networks (in particular, simulating ATM networks) is not defined very well because ATM is more than a specific networking solution. An ATM interface must conform to certain rules, but these rules leave a substantial amount of freedom with respect to the actual implementation. In particular, the number of parameters characterizing an ATM switch is large and some of these parameters, like the call admission policy and the buffering strategy, are not easily representable by simple values or symbols with a universal meaning. If we want to develop a general model of ATM networks, we have to find a way of representing all these parameters without restricting their possible ranges and without oversimplifying the model beyond a reasonable level.

Another important issue is the granularity of the model. The combination of the high transmission rate of ATM networks and the relatively small cell size results in a large number of events needed in order to simulate the network behavior for a non-trivial amount of real time. The diversity of traffic patterns in a reasonably-sized ATM network adds to the problem by extending the amount of real time over which the model should be run to produce meaningful results. Consequently, simulating ATM networks is a time-consuming endeavor and the efficiency of the model becomes a serious concern.

In this paper, we discuss a simple generic model of ATM networks. The model has been programmed in SMURPH [3] and operates at the cell level. Its objective is to mirror as closely as possible the timing of all cell transmission/reception events occurring in the corresponding physical network. The presented simulator is the first leg of a much larger project aimed at the development of a parallel simulation facility for ATM networks.

*Supported in part by NSERC Grant No. OGP9110. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. email: dobo@cs.ualberta.ca.

†Supported in part by NSERC Grant No. OGP9183. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. email: pawel@cs.ualberta.ca.

2 Model assumptions and simplifications

The intended expressing power of our model should make it possible to describe meshes of ATM switches of different types. The type of a switch is characterized by its connectivity, i.e., the number of input and output ports (which are assumed to be the same), its signaling rules (which in our model reduce to the connection setup protocol and call admission policy), and the buffering/policing scheme describing how and where the incoming cells are buffered and what happens when the switch runs out of buffer space. Every output port of a switch is assigned a definite transmission rate. This rate is assumed to be an attribute of the (unidirectional) channel (link) connecting the output port to the corresponding input port of another switch. Each ATM channel is represented by a pair of unidirectional links connecting the same pair of switches but in the opposite directions.

Besides switches, the network consists of end-nodes representing the hosts interfaced to the communication subnet.[1] Each end-node is connected via an ATM channel (i.e., a pair of links) to one switch and looks like a trivial switch with a single pair of ports. The topology of the modeled network doesn't change during simulation. Moreover, the structure of *virtual paths* (VP's) is also assumed to be static over the simulated time interval. Consequently, the model doesn't handle VPI switching. For every destination (end-node) reachable from a given switch, the switch maintains a number (possibly larger than one) of routes (output ports) via which the destination can be reached. The decision as to which route should actually be followed is made during call-setup processing, based on the call admission algorithm associated with the switch. In particular, if no potential route offers enough free bandwidth for the requested connection, the call is rejected.

All traffic in the network, including the data traffic and signaling (call setup) messages originates at end-nodes and is ultimately addressed to end-nodes. Various call-setup messages are processed internally at intermediate switches. A call-setup request may be rejected before it reaches the destination end-node. In such a case, the originating end-node is notified with a pertinent rejection message. It is assumed that all signaling messages exchanged during call setup are single-cell messages.[2] The call-setup algorithm is in principle flexible and can be modified on a per-switch basis.[3]

One simplification that reduces the amount of processing required for every cell arriving at the switch, without affecting the timing of all relevant events, is the elimination of explicit VCI switching. For every connection to be established, the VCI (*virtual circuit identifier*) is selected globally from a central pool of available identifiers and used to tag all cells carrying traffic related to the connection. When the connection is set up, every switch along its path sets up an entry in its internal table, which associates ports with the VCI. As the VCI is global, there is no need to change it when a cell is transferred from one switch to another. The same number is used to identify the connection at every switch along the path.

3 Generic switch structure

SMURPH is an object-oriented protocol specification language based on C++, built on top of an event-driven simulator providing a virtual environment for executing protocols expressed in that language [3]. A natural way to define switches (and other data types) that can be re-specified in their several customized versions is to take advantage of the type inheritance mechanism of C++. In particular, the most generic node of an ATM network can be defined in SMURPH in the following way:

[1] According to the OSI terminology [2].

[2] This simplification has been adopted for the presentation only and it can be easily removed.

[3] Its version discussed here represents a subset of the Q.93B standard [1].

```
station ATMNode {
  Port **IPorts, **OPorts;
  int NPorts;
  void setup (int np) {
    NPorts = np; IPorts = new Port* [NPorts]; OPorts = new Port* [NPorts];
    for (int i = 0; i < NPorts; i++) {
      IPorts [i] = NULL; OPorts [i] = NULL;
    }
  };
};
```

Type `ATMNode` covers all station types in our network, i.e., ATM switches as well as end-nodes. The two arrays of `Port` pointers represent collections of input (`IPorts`) and output (`OPorts`) ports interfacing the node to the network. The total number of port pairs is given by `NPorts`, which is set by the `setup` method when the station is created. Note that for an end-node station, `NPorts` is equal 1 and each of the port arrays consist of a single element.

The `setup` method of `ATMNode` will be called when the station is created. The only parameter of the method will then give the node's connectivity. Note that the `setup` method doesn't create the ports, but it allocates memory for the arrays and fills their entries with special values representing nothing. The ports will be created and the arrays filled when the station is configured into a network.

An ATM switch is defined as a subtype of `ATMNode`. There is no such thing as "**the** ATM switch" (several versions of ATM switches are available commercially [5] and much more switch types will be developed in the future); therefore, we cannot describe all possible ATM switches in a single type. It makes sense, however, to encapsulate the common properties of all ATM switches into one base type that will be used to derive other switch types representing specific switch models. This base type can be defined as follows:

```
station Switch : ATMNode {
  short VCITable [MAXCONNECTIONS] [2];
  Mailbox **Arrivals;
  virtual void csetup (Cell*, int) { };
  virtual void enter (Cell*, int) { };
  virtual CBuf *acquire (int) { return NULL; };
  void setup (int np) {
    ATMNode::setup (np);
    Arrivals = new Mailbox* [np];
    for (int i = 0; i < MAXCONNECTIONS; i++)
      VCITable [i][0] = VCITable [i][1] = NONE;
    for (i = 0; i < np; i++) Arrivals [i] = create Mailbox;
  };
};
```

For each (global) VCI representing a connection path passing through the switch, array `VCITable` gives a pair of port indexes. A cell tagged with a given VCI and arriving on the input port described by one element of the corresponding pair is to be relayed on the output port represented by the other element of the pair. This way the pair of indexes describes a two-way connection (note that all ATM connections are bidirectional).

`Arrivals` is an array of `Mailbox` pointers, each mailbox associated with the corresponding output port. Mailboxes are SMURPH objects used to communicate processes. One process can deposit a signal into a mailbox and this signal can be perceived as an interrupt by another process. In our model, a signal appearing in the mailbox number i tells that an outgoing cell directed to the output port

number i is available for transmission. Such a signal will be directed to the process servicing the corresponding output port.

The setup method of `Switch` invokes the setup method of its supertype and builds the two arrays. The mailboxes, being SMURPH objects, are constructed with the special operator `create`.

The other three methods are placeholders for the switch's functional modules that must be provided individually for each specific switch type. They are declared as virtual because they will be re-specified in subtypes of `Switch`. Method `csetup` will be called upon the arrival of a signaling cell; its role is to carry out the connection setup protocol. The first argument points to the cell itself and the second gives the index of the input port on which the cell has arrived. Method `enter` is used to deposit outgoing cells in the switch's buffer storage. The first argument identifies the cell to be deposited and the second gives the index of the output port on which the cell should be transmitted. The third virtual method (`acquire`) empties the buffer storage by removing from it the buffer containing the next cell to be transmitted on the output port indicated by the argument. Thus, `enter` and `acquire` hide the implementation of the buffering strategy used by the switch.

The functional schematic of an ATM switch can be programmed and discussed without assuming any specific implementation of the three virtual methods. In SMURPH, the behavior of a network station is described by a collection of *processes*. Below we list the complete specification of the *input* process. A separate instance of this process services every input port.

```
process Input (Switch) {
  Port *IPort; int PIndx;
  void setup (int pn) { IPort = S->IPorts [PIndx = pn]; };
  states {WaitCell, Enter};
  perform { int op;
   state WaitCell:
    IPort->wait (BOT, Enter);
   state Enter:
    if (TheCell->Type != DTC)
      S->csetup (TheCell, PIndx);
    else {
      if ((op = S->VCITable [TheCell->VCI][0]) == PIndx)
        op = S->VCITable [TheCell->VCI][1];
      if (op != NONE) S->enter (TheCell, op);
    }
    proceed WaitCell;
  };
};
```

When the process is created, its setup method assigns the index of the input port to be serviced by the process to its local attribute `PIndx` and the pointer to the actual port to `IPort` (`S` is a standard attribute pointing to the station at which the process is running). The process code (starting with the `perform` keyword) describes a two-state machine. Its execution starts in the first state (`WaitCell`) in which the process waits for a cell arrival on the input port (event `BOT`). When this happens, the process transits to state `Enter` where it examines the cell type. If the cell is not a data cell (meaning that it is a signaling cell), the process calls the `csetup` method of the switch. Otherwise, based on the `VCI` attribute of the cell, the cell is "entered" to the buffer storage and tagged with the index of the output port.[4] Having completed the processing of one incoming cell, the process moves back to state `WaitCell` (operation `proceed`) to await another cell arrival.

[4]Note that data cells with incorrect VCI's (unknown by the switch) are ignored.

The other end of the switch is serviced by a collection of *output* processes, one output process associated with one output port.

```
process Output (Switch) {
  Port *OPort; int PIndx; CBuf *SBuf; Mailbox *Arrival;
  void setup (int pn) {
    OPort = S->OPorts [PIndx = pn]; Arrival = S->Arrivals [PIndx];
  };
  states {Acquire, XDone};
  perform {
   state Acquire:
    if ((SBuf = S->acquire (PIndx)) == NULL)
      Arrival->wait (NEWITEM, Acquire);
    else
      OPort->transmit (SBuf->cell (), XDone);
   state XDone:
    OPort->stop (); SBuf->free ();
    proceed Acquire;
  };
};
```

The process starts in state `Acquire` where it executes the `acquire` method of the switch, trying to get a ready cell directed to the output port serviced by the process. If this operation fails (`acquire` returns `NULL`), the process awaits a signal on the mailbox associated with the port (event `NEWITEM`), indicating the arrival of a cell directed to the port, and then tries again. Note that successful `acquire` returns a pointer to the cell buffer. To get hold of the cell, the process executes the `cell` method of the buffer (see the next section). The cell is transmitted on the output port (operation `transmit`) and when this operation is complete the process transits to state `XDone`. In that state, the process terminates the cell transmission, frees the buffer,[5] and proceeds back to state `Acquire` to take care of another outgoing cell.

4 Cells and buffers

SMURPH offers built-in data types for representing both high-level messages exchanged by communicating hosts and low-level frames (packets) into which this messages must be turned before they can be submitted to the network. A message is represented in our model by the following structure:

```
message ATMMessage {
  int VCI, Type, SeqNum;
  FLAGS Attributes;
};
```

As we mentioned earlier, VCI's are selected globally and they uniquely represent connections in the domain of the entire network. The `VCI` attribute of `ATMMessage` identifies the connection (session) to which the message belongs. Attribute `Type` indicates the message type and is used to identify various kinds of signaling messages and to tell data messages from signaling messages. `SeqNum` is the serial number of the message; its role is to enable the recipient to detect lost messages. `Attributes` is a collection of binary flags that describe some properties of the message. For example, within the

[5] The reason why we have to distinguish between cells and cell buffers at this level is the need to be able to implement correctly the operation of freeing the storage occupied by an outgoing cell.

class of "data" messages, some messages may actually carry the data (e.g., a fragment of a file being transferred), while some others may represent acknowledgements.[6] An `Attribute` flag can be used to mark a data message as special, so that it can be identified and processed in a special way.

Some attributes of messages are implicit and their automatic declarations are invisible. Examples of such attributes are: `Length` (message length in bits) and `Receiver` (the identifier of the end-node to which the message is addressed). Global properties of messages and their general characteristics (e.g., length distribution, arrival process, delay sensitivity) are described in *traffic patterns*, which are SMURPH objects modeling network customers. The union of all such customers is represented by a single compound object known as the `Client`. The operation of converting messages into cells (the AAL of ATM [4]) is implemented in the traffic patterns. Consequently, a traffic generator in our model supplies the network directly with cells to be transmitted to their destinations. A single traffic pattern can have a number of sessions (connections) active at any given time. Every such a session operates at two ends: the *source* and the *destination* of the data stream generated during the session. In some cases (e.g., acknowledgements for file transfers), it is legal for a destination to send a message addressed to the source. No multicast traffic is implemented at present.

Individual ATM cells are described by the following data type:

```
packet Cell {
  int VCI, Type, SeqNum;
  FLAGS Attributes;
  void setup (ATMMessage *m) {
    VCI = m -> VCI;
    Type = m -> Type;
    SeqNum = m -> SeqNum ++;
    Attributes = m -> Attributes;
  };
};
```

All attributes are inherited from `ATMMessage`. Every cell is assumed to have been derived from some message. As a byproduct of the standard SMURPH operation of acquiring a *packet* (a `Cell`) from a *message* (an `ATMMessage`), the setup method of the packet is called with the argument pointing to the message from which the packet is acquired. Note that with every cell acquisition, the sequence number of the message (`SeqNum`) is incremented by one. Therefore, for every two consecutive cells derived from the same message, the serial number of the second cell is equal to the number of the first cell+1.

A cell arriving at a switch is usually buffered before it can be retransmitted. Once a connection has been established, the buffering policy is the most important factor affecting the performance of the switch. The complete structure of the buffer storage depends on the buffering policy and its full specification is only possible in the context of a specific switch type. We can think, however, of a generic cell buffer type which can be defined as follows:

```
class CBuf {
  char CellHolder [sizeof (Cell)];
 public:
  void load (Cell *c) { *((Cell*) CellHolder) = *c; };
  Cell *cell () { return (Cell*) CellHolder; };
  virtual void free () { };
};
```

[6] Note that these acknowledgements are not signaling messages in the ATM sense.

Attribute `CellHolder` is just an array of bytes capable of accommodating a cell. The first two methods are used to access this storage: `load` loads the buffer with the indicated cell and `cell` returns a pointer to the cell currently stored in the buffer. The last method should be called to deallocate the buffer (see the previous section). Its contents depend on the organization of the buffer pool at the switch; therefore, this method is virtual.

Type `CBuf` is generic and it must be extended to be useful. A complete buffer type will typically specify additional attributes, e.g., links needed to put the buffers into lists, sets, or more sophisticated hierarchies. These attributes will be referenced by the `enter` method of the switch and the `free` method of the buffer.

5 End-nodes

End-nodes, representing hosts in our model, have only one pair of ports each and do no switching. As the process of message arrival and the entire logical structure of a communication session (including the operation of turning messages into cells) are described in traffic patterns, the behavior of an end-node is simple and it can be completely described in a generic way. Below we list the definition of the end-node type.

```
station EndNode : ATMNode {
  Cell CurrentCell;
  TIME CTimeout;
  void setup () { ATMNode::setup (1); };
};
```

`EndNode` is a station type descending from `ATMNode`. The only action performed by the setup method of `EndNode` is to call the setup method of the supertype with the argument equal 1, to indicate that the station should be equipped with a single pair of ports.

Attribute `CurrentCell` is used as a temporary storage for the current cell acquired from the `Client` to be transmitted by the node. `CTimeout` is a constant (which may be different for different end-nodes) representing the amount of time after which an unacknowledged connection request is presumed lost. The value of this constant is set by the module responsible for building the network and interfacing all its nodes.

Every end-node runs two processes. one of them, called the *source* process acquires cells for transmission and sends them on the outgoing link, while the other (the *destination* process) absorbs received cells. The source process is defined as follows:

```
process Source (EndNode) {
  Port *OPort; Cell *CurrentCell;
  void setup () { OPort = S->OPorts [0]; CurrentCell = &(S->CurrentCell); };
  states {NewCell, XDone};
  perform {
    state NewCell:
      if (Client->getPacket (CurrentCell, PayloadSize, PayloadSize,
       CellSize - PayloadSize))
        OPort->transmit (CurrentCell, XDone);
      else
        Client->wait (ARRIVAL, NewCell);
    state XDone:
      OPort->stop ();
      CurrentCell->release ();
```

```
        proceed NewCell;
    };
};
```

As is customary with SMURPH processes, the setup method of `Source` assigns pointers to the station attributes used by the process to its local variables. In its first state, the process tries to acquire a cell for transmission (operation `getPacket` addressed to the `Client`). The cell will be stored in the structure pointed to by the first argument of `getPacket`. The remaining arguments describe the *packetization* of the cell, i.e., the minimum and maximum size of its payload and the size of the header, i.e., the portion that doesn't count to the payload but contributes to the total length of the cell. ATM cells are fixed in size; thus, the minimum size of their payload is equal to the maximum size.[7]

If a cell is available, `getPacket` returns non-zero. This means that an outgoing cell has been stored in `CurrentCell`. In such a case, the process initiates its transmission (operation `transmit` addressed to the output port) and transits to state `XDone` when the entire cell has been transmitted. Otherwise (no cell awaiting transmission), the process suspends itself (operation `wait` addressed to the `Client`) awaiting the standard event `ARRIVAL` triggered by the arrival of a `Client` message to the end-node. When it happens, the process will wake up in state `NewCell` where it will re-execute `getPacket`, this time successfully.

In state `XDone`, the process terminates the transmission and performs `release` on the cell. This standard operation is perceivable by the traffic generation process and its role is to notify the `Client` that the node is done with the cell. One purpose of `release` is to update various standard performance measures calculated automatically by the `Client`. Having completed the processing of one cell, the process moves back to state `NewCell` to take care of another one.

The other process that runs at `EndNode` has the following structure:

```
process Destination (EndNode) {
  int MySId; Port *IPort;
  void setup () { MySId = S->getId (); IPort = S->IPorts [0]; };
  states {WaitCell, Receive};
  perform {
    state WaitCell:
      IPort->wait (EOT, Receive);
    state Receive:
      Client->receive (TheCell);
      proceed WaitCell;
  };
};
```

Attribute `MySId` is set by the setup method to the `Id` (address) of the station (end-node) owning the process. `IPort` is a local pointer to the input port of the end-node.

The process starts in state `WaitCell` where it issues a standard wait request (for event `EOT`) to the input port. With this request, the process will be awakened in state `Receiver` when the last bit of a cell is perceived by the port. In state `Receive`, the process executes the standard `receive` operation of the `Client` (to notify the traffic generator about the reception of a new cell) and moves back to state `WaitCell` to await another cell reception.

[7] `PayloadSize` is a constant defined as 352 (bits). Similarly, `CellSize` is set to 424 bits.

6 A switch example

In this section we discuss a sample switch architecture with a specific call admission policy and buffering strategy. The switch has a pool of buffers associated with every output ports. A data cell arriving at the switch is stored at the end of the FIFO list of cells destined for a given output port. If no buffer space is available at the port, the last cell from the list is dropped. More specifically, the buffering strategy deals with two types of cells called "red" and "green.[8]" Red cells are considered less critical than green cells and, if there is no room to accommodate a new outgoing cell into a port queue, the switch will try to drop a red cell first. Only if no such cell is available will the switch consider dropping a green cell.

Below we list the type declaration of the cell buffer for our sample switch. This type is an extension of type `CBuf` discussed in section 4.

```
class FifoItem : public CBuf {
  friend class Fifo;
  FifoItem *prev, *next;
  Fifo *Pool;
  FifoItem (Cell *cl) {
    load (cl);
    prev = next = NULL; Pool = NULL;
  };
  void free () { Pool->Used--; delete this; };
};
```

The extension consists of three pointers; two of them (`prev` and `next`) provide bidirectional list links needed to implement the FIFO queue, and the third (`Pool`) points to the buffer pool (FIFO queue) to which the buffer is linked. The standard constructor builds a buffer from the specified cell and initializes the three pointers to `NULL`. The `free` method deallocates the memory assigned to the buffer structure and decrements the usage count of the buffer pool.

Class `Fifo`, mentioned as a "friend" of `FifoItem` in the above declaration, represents the actual storage associated with a given output port. Its complete structure looks as follows:

```
class Fifo {
  FifoItem *Head, *Tail;
  int MaxSize, Used;
  void discard (FifoItem *f) {
    if (f->prev != NULL) f->prev->next = f->next;
    if (f->next != NULL) f->next->prev = f->prev;
    if (f == Tail) Tail = f->prev;
    if (f == Head) Head = f->next;
    f->free ();
  };
 public:
  Fifo (int s) { Head = Tail = NULL; MaxSize = s; Used = 0; };
  CBuf *head () {
    CBuf *h;
    if (Head) {
      h = Head;
      if ((Head = Head->next) != NULL) Head->prev = NULL;
```

[8] In the model, the red/green cell type is selected by an `Attribute` flag. In reality, the ATM cell header carries a *priority* filed which is used for this purpose.

```
      return h;
    } else
      return NULL;
  };
  void add (Cell *cl) {
    FifoItem *f;
    if (Used == MaxSize) {
      for (f = Tail; f != NULL; f = f->prev)
        if (flagSet (f->cell () . Attributes, RED) break;
      discard (f ? f : Tail);
    }
    f = new FifoItem (cl); f->Pool = this;
    if (Head) {
      Tail->next = f;
      f->prev = Tail;
      Tail = f;
    } else
      Tail = Head = f;
    Used++;
  };
};
```

The two pointer attributes `Head` and `Tail` point to the first and the last cell buffer in the queue. `MaxSize` (set by the constructor) gives the maximum capacity of the buffer pool and `Used` tells how many cells are currently stored in the queue. Method `head` removes from the queue the first cell buffer and returns the pointer to this buffer. Note that the buffer is not formally deallocated until the output process responsible for transmitting the cell has accomplished its task and executed `free` for the cell buffer (see section 3).

A process willing to deposit a cell in the FIFO storage should call `add` specifying the cell pointer as the argument. If the buffer pool happens to be full (`Used == MaxSize`), the method will discard the last-queued red cell, or simply the last cell if no red cell can be found.[9] Then, `add` creates a new `FifoItem` containing the cell, appends it at the end of the FIFO list, and increments the used count by one.

Now it is time to look at the declaration of the switch type.

```
station ASwitch : Switch {
  CReqPool *Connections; Fifo **Buffers; TIME CTimeout;
  int *RouteSize, **Routes, *Used, *Bandwidth;
  void setup (int np) {
    Switch::setup (np);
    Connections = new CReqPool;  Buffers = new Fifo* [np];
    RouteSize = new int [np];    Routes = new int* [np];
    Used = new int [np];         Bandwidth = new int [np];
    for (int i = 0; i < np; i++) Used [i] = 0;
  };
  void csetup (Cell*, int);
  int admit (Cell*);
  void cancel (Cell*, int);
  void enter (Cell *cl, int BIndx) {
```

[9] Note that `discard` calls `free` and, consequently, it decrements `Used` by one.

```
    Buffers [BIndx] -> add (cl);
    Arrivals [BIndx] -> put ();
  };
  CBuf *acquire (int BIndx) { return Buffers [BIndx] -> head (); };
};
```

The only type we don't know at the moment is `CReqPool` describing a collection of pending connection requests currently processed by the switch. Besides `Connections`, the connection setup procedure uses the four integer arrays, attribute `CTimeout`, and three methods: `csetup`, `admit`, and `cancel`. The connection setup procedure, as well as the role of all these items, will be discussed in the next section.

Array `Buffers` represents the collections of FIFO buffers associated with individual output ports. Although this array is created by the setup method of `ASwitch`, the operation of filling it (as well as the other arrays unfilled by the setup method) will be carried out by the module responsible for building the network and assigning specific configuration parameters to the individual switches and end-nodes.

Type `ASwitch` redefines the three virtual methods of `Switch` that describe its buffering strategy (`enter` and `acquire`) and the connection setup algorithm (`csetup`). Method `enter` is invoked by the input process (see section 3) for every incoming cell that is to be relayed via one of the output ports. The simple operation performed by this method is to `add` the new cell to the FIFO buffer associated with the port. Similarly, `acquire` extracts the first outgoing cell (the `head`) from the buffer corresponding to the indicated output port.

7 Connection setup procedure

The connection setup procedure implemented in our model can be viewed as a simplified[10] variant of the Q.93B signaling protocol. Although the `csetup` method (which is the heart of this procedure) is specified separately for every switch type, it is assumed that all switches obey the same general rules and they can consistently exchange connection requests among themselves. Our subset of signaling messages does not cover the operations like host identification (the network configuration is static) or multicast connections. Besides, we assume that all signaling messages are single-cell. This way we get rid of the reassembly problems at the signaling level which, although not very challenging from the programmer's perspective, would obscure our presentation and make it much longer.

At any moment a single switch can handle a number of connection requests, each request possibly being in a different stage of processing. The set of such pending connection requests at a given switch is described by a list of request descriptors, each descriptor represented by the following data structure:

```
class CReq {
 public:
  int Bandwidth, VCI, IPIndx, OPIndx, Status;
  ReSender *RS;
  CReq *next;
  CReq (int vc, int bd, int ip, int op, ReSender *r) {
    Bandwidth = bd; VCI = vc; IPIndx = ip; OPIndx = op; RS = r;
    Status = PENDING;
  };
};
```

[10] For the sake of presentation.

For simplicity, we assume that the bandwidth requirements specified in a connection request are described by a single number stored in `Bandwidth`. The `SeqNum` field in the cell structure (see section 4) is used to carry this information in the signaling cell.[11] Attributes `VCI`, `IPIndx`, and `OPIndx` stand for the (global) VCI of the connection, the index of the input port (on which the request has arrived), and the index of the output port (via which the connection will proceed to the destination). The current state of the connection is stored in attribute `Status`. The legitimate values of this attribute are: `PENDING`, `ESTABLISHED`, and `CLEARING`. Attribute `RS` is a process pointer. Occasionally, there will be a special process (of type `ReSender`) associated with a connection request, responsible for periodic re-sending of a signaling cell to the next switch along the connection's path, until a confirmation is received. The role of the `next` pointer is to link multiple request descriptors into a list—the so called *request pool* representing all connections currently handled by the switch. The request pool is described by the following class:

```
class CReqPool {
  CReq *Head;
 public:
  CReqPool () { Head = NULL; };
  void store (int vc, int bd, int ip, int op, ReSender *r) {
    CReq *cr = new CReq (vc, bd, ip, op, r);
    cr->next = Head; Head = cr;
  };
  CReq *find (int vc, int op) {
    for (CReq *cr = Head; cr != NULL; cr = cr->next)
      if (cr->VCI == vc && cr->OPIndx == op) return cr;
    return NULL;
  };
  CReq *pending (int vc, int ip) {
    for (CReq *cr = Head; cr != NULL; cr = cr->next)
      if (cr->VCI == vc && cr->IPIndx == ip &&
         cr->Status != CLEARING) return cr;
    return NULL;
  };
  void purge (int vc) {
    CReq *cr, *cq;
    for (cr = Head, cq = NULL; cr != NULL; cq = cr, cr = cr->next)
      if (cr->VCI == vc) {
        if (cq == NULL)
          Head = cr->next;
        else
          cq->next = cr->next;
        delete cr;
        if ((cr = cq->next) == NULL) break;
      }
  };
};
```

An empty request pool is characterized by the `Head` attribute containing `NULL`. Method `store` creates a new request description (parametrized by the argument list) and adds it to the pool.[12] Method `find` locates the request matching the specified VCI number and the output port index.

[11] More complex bandwidth requirements can be implemented by interpreting this number as an index pointing to a more elaborate description.

[12] The ordering of the descriptors in the list is immaterial.

With `pending`, one can find the request that is either *pending* or *established* and matches the given VCI number and the input port index. Note that the last two methods return `NULL` if the descriptor cannot be found. The purpose of `purge` is to remove the indicated descriptor from the pool.

The methods of `CReqPool` are used by `csetup` which is the only module performing operations on request pools. It may not be clear why a request descriptor must be identified by a pair of values: the VCI number and a port index. It would seem that the global VCI numbers provide a sufficient means of identifying connection requests uniquely. Unfortunately, it may happen that a connection request appears cleared to the originating party, while its remnants are still present in the network. The connection setup algorithm guarantees that these remnants will eventually disappear, but in the meantime, if the same VCI number is assigned to another connection request, it is possible that two or even more request descriptors in the same pool will be tagged with the same VCI. It turns out that using port numbers together with the VCI numbers is a sufficient protection against confusion that otherwise might have arisen in such situations.[13]

A connection setup request originates at an end-node and propagates through the network building a path from the source node to the destination. A switch receiving such a request has to decide whether to admit the call and, if the decision is affirmative, which outgoing channel should be used as the next link of the connection path. It is conceivable that the same destination can be reached via several output ports, but once the connection has been set up, the output port must be determined and fixed. In our model, every `ASwitch` maintains a two-dimensional table called `Routes`, which for every output port gives the list of destinations that can be reached via that port. The decision as to whether a call should be accepted or rejected is based on the availability of bandwidth. Every output port has a fixed number of *bandwidth units* (array `Bandwidth`) associated with it. Every connection setup request specifies the number of bandwidth units required for the connection (attribute `SeqNum` of the connection request cell). When a new connection is accepted and directed via a given output port, the used bandwidth of that port (array `Used`) is augmented by the number of units required for the connection. A new connection is only accepted if there exists an output port offering a route to the destination, whose unused bandwidth is at least as large as the requested bandwidth. Although all connections are in fact bidirectional, we assume for simplicity that the reverse bandwidth is zero (i.e., the amount of traffic going from the destination to the source is insignificant and it doesn't occupy any bandwidth resources of the switch).

All the above operations, i.e., the route selection, admission decision, and bandwidth allocation, are performed by the following method of `ASwitch`:

```
int ASwitch::admit (Cell *sc) {
  int i, j, op, ub, tb;
  for (ub = 0, i = 0; i < NPorts; i++)
    if (Connections->find (sc->VCI, i) == NULL && (tb = Bandwidth [i] - Used [i]) > ub)
      for (j = 0; j < RouteSize [i]; j++)
       if (Routes [i] [j] == sc->Receiver) { ub = tb; op = i; break; }
  if (ub < sc->SeqNum) return NONE; else {
    Used [op] += sc->SeqNum;
    return op;
  }
};
```

The method is called with a pointer to the connection request cell passed as the argument. The

[13] In a real network, a connection setup message is tagged with an index, which, in combination with the address of the originating node, provides a unique identifier of the request within the entire network.

outer `for` loop goes through all output ports and finds the one with the maximum available bandwidth (`Bandwidth[i]-Used[i]`) that offers a route to the destination. The first part of the `if` condition makes sure that the port has not been tried already for this connection. This way the switch can explore alternate paths ignoring the paths that were attempted, but the call failed somewhere up the link. The inner loop checks whether the destination (attribute `Receiver` of the signaling cell) occurs on the `Routes` list associated with the output port number `i`. If the bandwidth available at the selected port is less than the bandwidth indicated by the signaling cell, the method returns `NONE`, which indicates a failure. Otherwise, the method increases the used bandwidth of the port by the requested bandwidth of the new connection and returns the index of the selected output port.

The following method is called to release the bandwidth resources associated with a connection:

```
void ASwitch::cancel (Cell *sc, int ip) {
  CReq *cr;
  cr = Connections->pending (sc->VCI, ip);
  assert (cr, "cancel: connection not found");
  Used [cr->OPIndx] -= sc->SeqNum;
};
```

The method is called in response to a *disconnect request* received by the switch. The pointer to the signaling cell carrying this request, as well as the index of the input port of the connection, are passed to the method as the arguments. Note that the signaling cell is expected to carry the bandwidth requirements of the original connection request cell.

The connection setup procedure is implemented as a single method (`csetup`) which looks like a finite-state automaton driven by the type of a received signaling cell. The following types of signaling cells are recognized:

- `CST` call setup request,
- `ACK` call setup confirmation (when received indicates that the connection has been established),
- `CNK` call rejection,
- `DSC` disconnect request,
- `DAK` disconnect acknowledgement (when received means that the connection has been cleared).

We assume that a disconnect request can only originate at the calling party, i.e., the source end-node that requested the connection.

Occasionally, method `csetup` will spawn a process that will be periodically re-sending a signaling cell until a response is received by the switch. This is a natural way of implementing timeouts in SMURPH. The process taking care of this end is defined as follows:

```
process ReSender (ASwitch) {
  int OPIndx; CBuf SCBuf; TIME Timeout;
  void setup (Cell *c, int op, TIME tm) {
    SCBuf . load (c); OPIndx = op; Timeout = tm;
  };
  states {Send, Sleep};
  perform {
    state Send:
      S->enter (SCBuf . cell (), OPIndx);
      Timer->wait (Timeout, Send);
  };
};
```

When created, the process makes its private copy of the signaling cell to be sent on the indicated output port (attribute `OPIndx`) and stores it in `SCBuf`. The cell pointer, the output port index, and the timeout value are passed to the process upon creation via the arguments of the setup method. The process' code method consists of the single state `Send`. In this state, the process `enter`s the control cell to the switch (see section 3) directing it to the indicated output port, then waits for `Timeout` time units, and resumes execution in state `Send` to issue another copy of the cell. The process will be performing these simple operations indefinitely, until it is killed by `csetup`, when the method detects the arrival of the awaited response.

The connection setup method is relatively long, but its different fragments, representing responses to the different signaling cell types, can be discussed independently. Below we list the complete code of `csetup`.

```
void ASwitch::csetup (Cell *sc, int ip) {
  int i, op; ReSender *rs; CReq *cr;
  switch (sc->Type) {
    case CST:
      if (cr = Connections->pending (sc->VCI, ip)) {
        if (cr->Status == ESTABLISHED) {
          sc->Type = ACK; enter (sc, ip);
        }
      } else if ((op = admit (sc)) != NONE) {
        rs = create ReSender (sc, op, CTimeout);
        Connections->store (sc->VCI, sc->SeqNum, ip, op, rs);
      } else {
        sc->Type = CNK; enter (sc, ip);
      }
      return;
    case ACK:
      cr = Connections->find (sc->VCI, ip);
      if (cr && cr->Status == PENDING) {
        cr->RS->terminate (); cr->Status = ESTABLISHED; enter (sc, cr->IPIndx);
        VCITable [cr->VCI][0] = cr->OPIndx; VCITable [cr->VCI][1] = cr->IPIndx;
      }
      return;
    case CNK:
      cr = Connections->find (sc->VCI, ip);
      assert (cr, "csetup: NAK for an unknown connection request");
      if (cr->Status == PENDING) {
        cancel (sc, cr->IPIndx); cr->RS->terminate ();
        cr->Status = CLEARING; sc->Type = DSC;
        cr->RS = create ReSender (sc, ip, CTimeout);
        if ((op = admit (sc)) != NONE) {
          sc->Type = CST;
          rs = create ReSender (sc, op, CTimeout);
          Connections->store (sc->VCI, sc->SeqNum, cr->IPIndx, op, rs);
          cr->IPIndx = NONE;
        } else {
          sc->Type = CNK; enter (sc, cr->IPIndx);
        }
      }
      return;
```

```
      case DAK:
        if ((cr = Connections->find (sc->VCI, ip)) != NULL && cr->Status == CLEARING) {
          cr->RS->terminate ();
          if (cr->IPIndx != NONE) enter (sc, cr->IPIndx);
          Connections->purge (sc->VCI);
        }
        return;
      case DSC:
        if (cr = Connections->pending (sc->VCI, ip)) {
          cancel (sc, ip);
          if (cr->Status == PENDING)
            cr->RS->terminate ();
          else
            VCITable [cr->VCI][0] = VCITable [cr->VCI][1] = NONE;
          cr->Status = CLEARING;
          cr->RS = create ReSender (sc, cr->OPIndx, CTimeout);
        } else {
          sc->Type = DAK; enter (sc, ip);
        }
        return;
    }
  };
```

The arguments of `csetup` are the signaling cell pointer and the index of the input port on which the cell has arrived. The first `case` segment is executed for a connection setup cell (type `CST`). The method first checks if a description of the connection request is already present in the request pool. If it happens to be the case and the connection is already established, the setup request represents a timeout reaction of the originator, which is still uncertain about the connection. Perhaps the confirmation cell hasn't reached the source yet or it has been lost. Thus, the cell type is changed to `ACK` (connection confirmation) and the cell (with all its other attributes intact) is sent up the link to the source. Note that the output port index for `enter` is equal to the index of the input port on which the cell has arrived.

If the connection has not been established yet, but its description is present in the pool, it means that the connection setup request is being processed by the switch, which is awaiting a response from a downstream neighbor. In such a case, the (duplicate) `CST` cell is simply ignored. The switch will send a confirmation upstream as soon as one arrives from the downstream neighbor.

If the arriving `CST` cell is actually the first indication of the connection (no description matching the request is present in the pool), the method executes `admit` to assign a route to the connection and check if there is enough free bandwidth available to accept it. If the value returned by `admit` (and assigned to `op`) is not `NONE`, the connection has been accepted and `op` gives the output port index. Then, `csetup` creates a `ReSender` process which will take care of propagating the `CST` request up the link (on the output port indicated by `op`) and adds the description of the connection request to the pool.

The last case to be considered is the situation when `admit` returns `NONE`, which means that the switch cannot accept the connection. In such a case, the method sends upstream a single `CNK` cell indicating to the originator that the call has been rejected. Note that nothing wrong will happen if the cell doesn't make to the source end-node (thus, there is no need to create a `ReSender` process for this purpose). The end-node will eventually time out and send another `CST` cell. Most likely, the node will do the same upon the reception of the `CNK` cell.

Now let us see what happens in `case ACK`, when the switch receives an `ACK` cell representing a positive confirmation arriving from downstream. First, the method locates the description of the connection setup request in the pool. The `ACK` signal makes sense if the request has its description in the pool and its status is *pending*; otherwise, it is ignored. The status of the request is changed to *established* and the `ReSender` process propagating the connection request (`CST`) down the link is terminated and discarded. The `ACK` cell is propagated upstream (operation `enter`) and the pair of ports representing the connection is registered in `VCITable` (see section 3). From now on, for as long as the connection is alive, the switch will know how to relay data cells transmitted over the connection's path.

The semantics of the other segments of `csetup` are easy to deduce from the code. Note that having received a rejection from downstream (`case CNK`) the method re-executes `admit` to try to select an alternative route. As the previous description of the *pending* connection request is still present in the pool, `admit` will not try to use the same route as before. Only if `admit` returns `NONE` in these new circumstances, is the call assumed to have failed and a reject message (a `CNK` cell) is sent upstream. The multiple descriptions resulting from alternative path selections for the same call do not accumulate in the pool indefinitely. The `purge` operation executed when a call is cleared removes all descriptions tagged with the VCI of the cleared connection.

8 Final remarks

The limited size of this paper did not allow us to present here the detailed implementation of the traffic generators. A traffic generator (or *traffic pattern*) is a special SMURPH object responsible for generating *messages* at end-nodes according to the dynamic specification of the arrival process, which may take the form of an interrupt-driven program. SMURPH offers simple built-in tools for organizing such programs in a concise and modular way. In particular, they can automatically split messages into cells (capturing the AAL of ATM), account for correlations between cell types, detect and diagnose lost cells, create feedback traffic at the destination, and make the source perceive this feedback. We have equipped our model with a number of standard traffic patterns, including video sessions and file transfers.

Even in its simplified version presented in this paper, our model proved quite useful for investigating some properties of realistic ATM networks under realistic loads. Its primary features can be stressed in the following two points:

- The model can mimic in an accurate way the timing of all the interesting events occurring at the cell-switching level.
- The model is highly modular, which makes it easy to add new switch types or traffic generators, or to change the functionality of the existing types.

The source code of our model, together with the SMURPH package and extensive documentation, is freely available from the authors.

References

[1] ATM User-Network Interface Specification, version 3.0. The ATM Forum, 1993.

[2] J. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, pages 1334–1340, Dec. 1983.

[3] W. Dobosiewicz and P. Gburzyński. SMURPH: An object oriented simulator for communication networks and protocols. In *Proceedings of MASCOTS'93, Tools Fair Presentation*, pages 351–352, Jan. 1993.

[4] Asynchronous Transfer Mode (ATM) and ATM Adaptation Layer (AAL) Protocols Generic Requirements. TA-NWT-001113, Bellcore, Issue 1, Aug. 1992.

[5] R. Rooholamini, V. Cherkassky, and M. Garver. Finding the right ATM switch for the market. *IEEE Computer*, 27(4):17–28, Apr. 1994.