Implementation of ATM Network Layer and Reliable ATM Adaptation Layer

John K. Edwards * Johannes E. Gehrke † Krzysztof Zmudzinski ‡

November 18, 1996

Contents

*edwards@cs.wisc.edu

†johannes@cs.wisc.edu

‡zmudzin@cs.wisc.edu

1 General System Structure

1.1 Terminology

There are three basic layers in our system. The data link layer, *DLL*, is a relatively simple wrapper to UDP ports, with the garbler possibly intervening, and is described in Section 2. The *ATM* layer provides the functionality of virtual circuit establishment and maintenance. *ATM* is detailed in Section 4. On top of *ATM*, there is *AAL7*. *AAL7* is a special case of an adaptation layer. The generic interface we provide between *ATM* and overlying adaptation layers is detailed in Section 3. Finally, *AAL7* (Section 5) is the particular adaptation layer we are implementing at this time.

We say that a thread is a *daemon thread* if it is created once at the beginning of a call to aal7_init() and is only terminated by a call to aal7_terminate(). These are threads that we will actually specify as Solaris daemon threads, meaning that if they are the only non-terminated threads left, the program will exit (of course, the upper level that called this may also choose to exit sooner, or to terminate ATM and AAL7)

In the remainder of the paper we will refer to atm *cells* and aal7 *packets* and so distinguish the two different levels of abstraction. We call all threads of control which make calls from the application layer to the AAL7 layer through the interface specified in aal7.h *application threads*.

All synchronizations of events are handled through a thread-safe, synchronized queue class. Temporal events are handled as follows: a message is sent to the timer object that enqueues a timer event into its local timer list. Whenever the timer has expired, the timer thread puts a message in the event queue of the respective thread.

1.2 Assignment of Tasks to Threads

There are four daemon threads in the atm layer (reader, writer, routing and VC table manager) and there is one daemon thread that is not associated with a particular layer: the timer thread.

Summarized, the reader thread reads incoming atm cells from the DLL and passes them on. The writer thread has an associated write queue. It picks lists of atm cells from the write queue and makes a call to the DLL to write them. The VC manager thread is responsible for establishing and tearing down connections. The routing thread is responsible for maintaining correct routing information that can be used by the VC manager thread.

- the reader thread

  The reader thread reads continuously at the port (through DLL); the read is a blocking read. It only wakes up whenever an atm cell arrives at the port, in which case one of the following two scenarios is the case:

  The cell header indicates it is a data cell. In this case the reader thread accesses the VC table and either updates the header in the cell according to information in the virtual channel table and puts it into the write queue to be written by the write thread, or if the cell is is being sent to a higher layer on this node (AAL7 for example), the reader thread makes an upcall to the higher layer to deliver the cell.

  The cell is a service cell. In this case the reader thread puts the cell either into the routing evenet queue or the VC table manager event queue.

- the routing thread

  It is the obligation of the routing thread to keep the routing table updated. It puts cells into the writer queue to maintain the connectivity information of the network. It receives cells from other nodes routing threads through the routing event queue (they were put there by the reader thread).

- the writer thread

  The writer thread takes cells from the write queue and sends them through a DLL call. It does not perform any header modifications.

- the VC table manager thread

  The VC table manager thread is responsible for tearing down connections and establishing connections. It maintains the virtual channel table. It will send many different kinds of messages to establish and maintain connections.

The general system structure is shown in figure 1.

1.3 Timer Management

We use one, centralized timer manager, similar to the one described in Peterson's book on pages 67-69. Both the *ATM* layer and *AAL7* layer schedule events using the same timer manager. *AAL7* protocol schedules a time-out event for each packet sent: if an acknowledgment is not received before some time-out time, an event is appended to the appropriate AAL7 connection event queue associated with a particular connection that causes the connection thread to to resend the packet. The virtual circuit signaling protocol uses time-out events during circuit establishment and teardown. After sending a message, a time-out event is scheduled to force re-transmission of the message if the request is not acknowledged before the time-out expires. The routing protocol uses the timer manager to schedule when to send echo ("pong" - "pong" rather than "ping" because "ping" connotes active sonar, our "pong" messages are more like passive sonar) cells to notify neighboring nodes

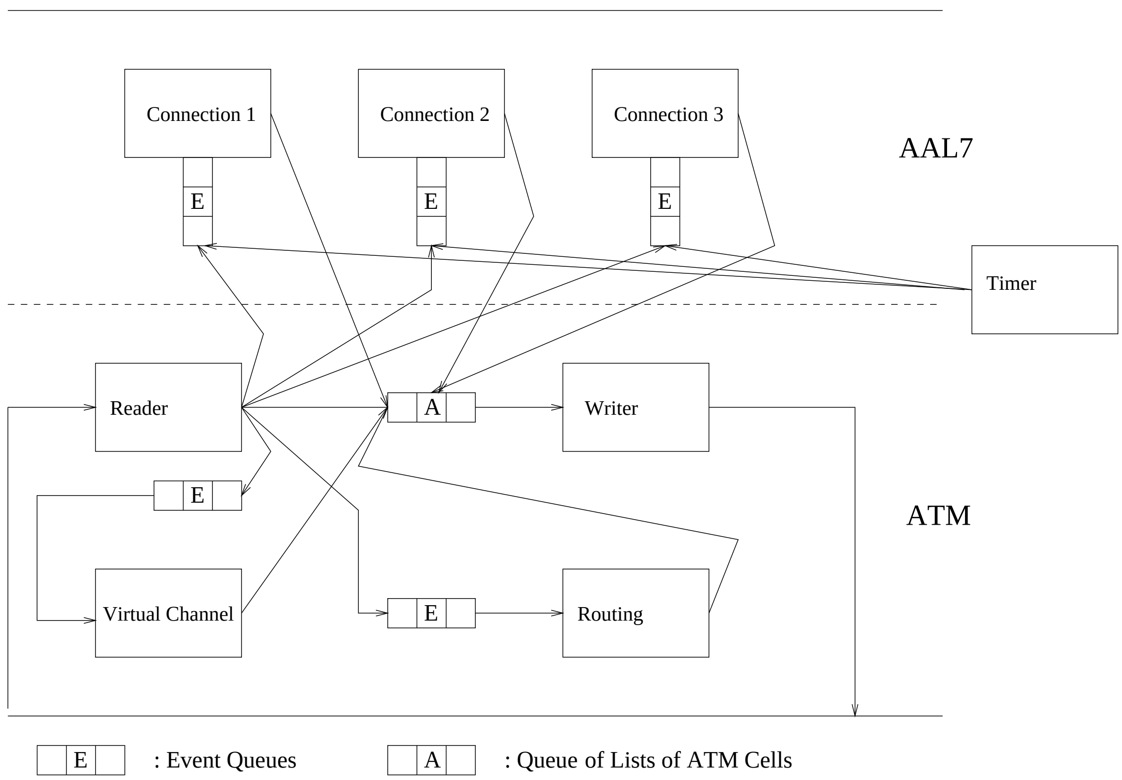

Figure 1: Overall system structure

about its upness. It also schedules time-outs for detection of nodes that went down: if a "pong" cell is not received from one of the neighbors of a particular node before the time-out expires, the neighbor is considered down. Lastly, the routing protocol schedules timer events for re-transmission of link-state cells if they are not acknowledged.

The timer manager consists of a linked list of time events structures sorted by expiration time. The `set_timer` method is used to add new time event to the queue and the `cancel_timer` method is used to remove an event from the queue if it is not needed any longer. The timer manager is protected by a mutex to protect consistency of the event queue.

Each time-out event structure has a pointer to an object of event_c class, a pointer to a synchronized queue of event_c class objects, expiration time and a special timer event handle. Expiration time of each event is relative to the previously scheduled event. The timer event handle is passed to the remove method to remove an event that is no longer needed. Every thread in the system (except for the writer and the reader threads) has a queue of events associated with it. When the time for a particular event expires, the event_c object is appended to the synchronized queue of event_c events associated with the thread on whose behalf the time event was scheduled. The same event queues are used to notify threads about other events in the system (e.g. cell arrived).

The timer manager thread will do timed condition waits on its event queue, waiting for messages to schedule or deschedule events. It will set the time to wait according to the next event in the delta list. If the delta list approach turns out to be inefficient, we will consider the timing wheels approach.

1.3.1 Interface

```
class event_c
{
 public:
  event_c();
  ~event_c();

  inline event_t get_event() { return event; }
  inline void* get_data() { return data; }

  inline void set_event(event_t ev) { event = ev; }
  inline void set_data(void* d) { data = d; }
 private:
  event_t event;
  void* data;
}
```

Every event in our project is encapsulated into an object of event_c class. Some events, like arrival events, have pointer to some generic data (ATM cells, or lists of ATM cells) set; in case of timer events this pointer is NULL.

```
class timer_manager_c {
public:
  timer_status_e set_timer(event_c *e, sync_queue_c<event_c*> *q,
   int ti, long& th);

  timer_status_e cancel_timer(long th);
private:
  sync_queue<event_s*> q;
}
```

The `set_timer()` method registers a timer event that will be called after `ti` 1/100ths of a second. It returns a handle `th` to the event in case it has to be removed from the delta list. If the event is not removed, the event `e` will be appended to the synchronized queue, `q`.

1.4 Checksum

In our design, we use the checksum algorithm from Peterson's book on pp. 109-110. The checksum computation is used for ATM service cells and for AAL7 packets. Both VC service cells and routing service cells are derived from service_atm_cell_c class which is a subclass of atm_cell_c with two additional methods: compute_checksum and check_checksum. Checksum is computed over the first 50 bytes of a service cell which means that we also include the header of a cell in the checksum computation. This is done to avoid complexity of operations on unaligned data. The checksum is stored in bytes 51 and 52 (53 is padding). For service cells, the header is not modified on each hop so including the header does not introduce any problems. For AAL7, the checksum is computed over the last 48 bytes of each cell in an AAL7 packet; since we calculate the checksum in the AAL7 layer, before the ATM layer has modified the 5 byte header field (which will be changed along the way.) we exclude the 5 byte cell headers from checksum computation. The checksum is stored in the last two bytes of the last cell (trailer) of an AAL7 packet.

Table 1: Synchronization

Shared Object	Threads accessing	Reason	Synchronization method
Routing Table	Routing thread VC Table Manager Application threads	correcting forward cells, make connections making connections	t_mutex (reader_writer?)
Virtual Circuit Table data	VC Table Manager Reader Thread AAL7 connection threads	manages table forward and deliver cells send cells, establish connections	t_mutex
Writer queue	Reader Thread Writer Thread AAL7 connection threads user threads	forwarding cells sends cells send cells send cells	synchronized queue
VC Table event queue	VC Table Manager Writer Thread Timer Thread Routing Thread	receives events deliver cells deliver events node down	synchronized queue
Free atm cell list	everybody!		synchronized queue

1.4.1 Checksum Algorithm

This is the standard IP checksum algorithm.

```
u_short
checksum(u_short *buf, int count)
{
  register u_long sum = 0;

  while (count--)
  {
    sum += *buf++;
    if (sum & 0xFFFF0000)
    {
      /* carry occured, so wrap around */
      sum &= 0xFFFF;
      sum++;
    }
  }
  return ~(sum & 0xFFFF);
}
```

This will of course be slightly modified to deal with a packet, which is a disjointed list of cells.

1.5 General Synchronization Issues

Most of our synchronization and protection of shared data is accomplished with synchronized queues. Below we provide a complete list of which threads access which data structures, and the synchronization methods used to access them:

Table 1 .

1.6 The synchronized queue class

Our main method to synchronize tasks between different threads and to have one conceptual place where threads can block is a synchronized queue class. Conceptually the queue consists of two queues: one queue for the arrival of events and one queue for the arrival of data. A thread that is waiting to dequeue the next element from the queue is waiting for an arrival in the event queue. So elements can be appended to the data-part of the queue without waking up any thread that is waiting for an arrival in the event queue. The event queue is the only place where a synchronized wait is possible, it is not possible to wait for data arrivals. In addition to blocking threads that want to dequeue elements, the queue is protected through a mutual exclusion, so that the insertion of elements in the data as well as the event part of the queue is thread-safe.

To ease implementation, we have implemented a template synchronized queue class. This template synchronized queue class is based on a template generic list class; due to the length of the report we did not include the header file for the generic list class into this report. From this template synchronized queue class and the generic list class, it is easy to build any types of queue, especially the special type of "double queue" (event and data queue) that we described in the preceeding paragraph. '

```
enum sync_queue_status_e {
  SYNC_QUEUE_OK,
  SYNC_QUEUE_FULL,
  SYNC_QUEUE_EMPTY
};

const int SYNC_QUEUE_UNBOUNDED = GENERIC_LIST_UNBOUNDED;

template<class T> class sync_queue_c : private generic_list_c<T>
{
public:

  sync_queue_c(int numberOfItemsi = SYNC_QUEUE_UNBOUNDED);
  ~sync_queue_c();

  sync_queue_status_e append(const T& item);

  sync_queue_status_e put_front(const T& item);

  sync_queue_status_e dequeue(T& item);
  sync_queue_status_e dequeue_blocking(T& item);

private:
  mutex_t queue_lockmut;
  cond_t  add_morecond;
  cond_t  get_morecond;
};

The {\tt append} method is used to add a member to the queue.
```

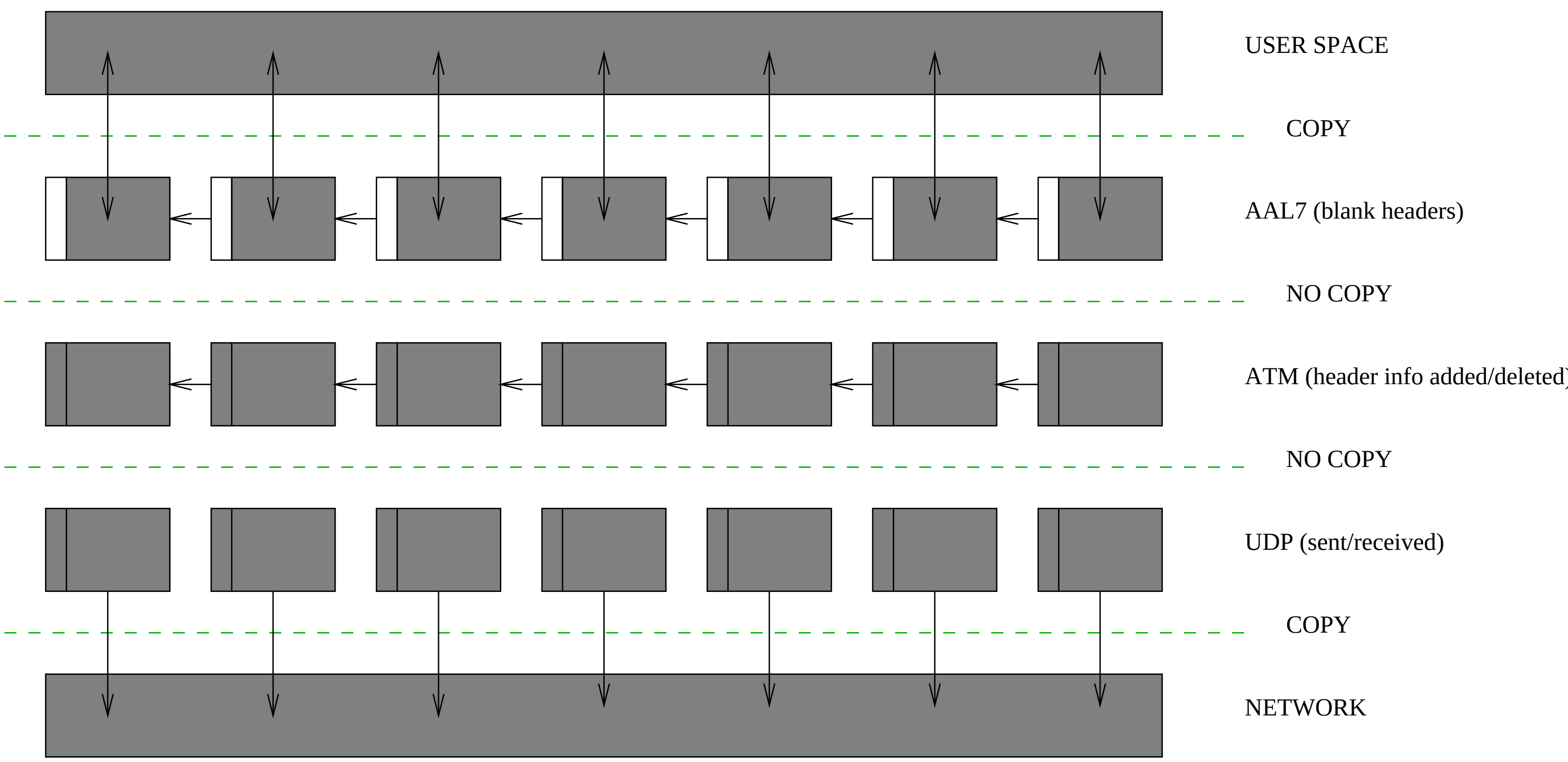

Figure 2: Illustration of handling data with no copies.

```
The {\tt put\_front} method prepends priority data to the queue.
This will be used for routing threads.  {\tt dequeue} is used to
remove an item from the queue or find out that the queue is empty.
If the calling thread should block until a {\tt item} is available,
{\tt dequeue\_blocking} is used.
```

1.7 Flow of Data

When the ATM layer is initialized, it allocates a global free cell list. This will help us avoid calling new to allocate cells on the fly. Whenever a thread needs to create a new atm cell, it will pull one off the list (calling new if the list is empty). Whenever a thread determines that a cell can be deleted, it is returned to the free cell list instead.

Whenever cells are passed from one thread to another, it is actually the pointer to the cell that is passed. Thus we do not copy the data. When AAL7 maintains packets or buffers, they are maintained as lists of atm cells with empty headers. When AAL7 delivers data to an Application, it copies it from the atm cells into the Applications buffer. When a packet is sent out, it is passed to ATM and eventually to the writer thread as a pointer to a list of free cells. This minimizes overhead of copying and passing cells. See Figure 2.

2 Data Link Layer

This is the lowest layer in our design. Its only function is to simulate a data link connection between nodes on the network. Each node is uniquely identified by the host number on which the node is "installed" and the port number used to send UDP datagrams; hence, each node has only one socket.

2.1 DLL Threads

There are two threads in the DLL: the reader thread and the writer thread. The reader thread blocks on the `recvfrom()` system call until a UDP datagram, which in our design corresponds to an ATM cell, arrives. Then, based on the data payload type, the cell is appended to one of the following queues:

- The virtual circuit manager queue if the payload type (`ATM_PT_VC` = 6) indicates a VC service cell.
- The DLL ATM cell queue if the payload type (`ATM_PT_DATA_CELL` = 0 or `ATM_PT_DATA_LAST_CELL` =1) indicates a data cell and the VC table indicates that the cell should be forwarded.
- An AAL7 connection queue (or other adaptation layer destination) if the cell is a data cell that the VC table indicates should be delivered up at this node.
- The router queue if the payload type (`ATM_PT_ROUTING` = 7) indicates a routing service cell.

By appending a cell to a queue we mean calling a method of the appropriate class that constructs an event and appends it to the queue that corresponds to either the VC manager, the VC table, or the routing protocol.

The writer thread blocks on the DLL ATM cell queue until a list of cells is present. Then, based on the virtual node number of the destination node, it looks up the actual address and port of that node and makes `sendto()` call for each cell.

2.2 DLL Data Structures

The DLL is responsible for sending ATM cells to correct hosts on which virtual nodes are installed. Therefore, it has to maintain a map that it uses to translate a virtual node number to the `sockaddr_in` structure. This map is an array of the following structures indexed by virtual node number.

```
typedef struct vn_to_ip_s {
  struct sockaddr_in s; // UDP/IP address information
  bool_e valid; // indicates whether node exists
} vn_to_ip_t;
```

The DLL is also responsible for receiving ATM cells. It has to refer to this map to find the virtual node number based on sockaddr_in structure of the host from which the cell was received. Right now this is a linear time lookup requiring a traversal of the entire map in the worst case.

Once a cell which is to be forwarded to another node is received by the reader thread, the reader thread refers to the VC table and after changing the header information of the cell, appends it to the DLL ATM cell queue in a form of one-cell list. At this point we think that we should not wait for more cells to arrive to make lists longer; only cells originating from the current node will be placed in longer lists. The DLL ATM cell queue is a synchronized queue of pointers to ATM cell lists:

```
sync_queue_c <atm_cell_list_c*> q;
```

2.2.1 DLL Interface

```
class dll_c {
 public:
  // initialize dll
  dll_status_e init(atm_host_t n, char* ncf);
  // writes port
  dll_status_e write(void);
  // reads port
  dll_status_e read(void);
  // close connetcion and clean up
  dll_status_e term(void);
...
```

The method `init()` builds the virtual-node-to-IP map based on the network configuration file. It also makes calls to the ATM routing table to build it based upon the configuration file and the virtual node number of the current node, it opens the UDP socket for this node and finally it initializes some private members used for statistics. The `write()` method starts the writer thread and the `read()` method starts the reader thread. The `term()` method closes the socket and "kills" the two threads.

2.3 Garbler

Since all reading to and writing from the underlying UDP sockets is done through the DLL, the integration of the Garbler will be transparent to the rest of the program.

3 The Interface Between an Adaptation Layer and the ATM Layer

In order to provide extensibility to other adaptation layers as well as to sharply define the different regions off programming responsibility, we have defined a standard interface between the ATM layer and the adaptation layers, of which AAL7 is obviously most important to us.

We divide the responsibilities between the two layers as follows: The ATM layer will provide the abilitity for any overlying adapatation layer to connect to the same adaptation layer on another node. The client will also be allowed to send a particular service number with which it wishes to connect. The server will be allowed to either accept the connection, or to reject the connection, returning an unsigned integer `error_code` that explains why it was rejected. The interpretation of the error code will be left entirely to the upper layer. The ATM layer also will allow the upper layer to send a list of ATM cells on any established circuit. This list will be of type atm_cell_list_c (or a derived class such as pin_sync_atm_cell_list_c which provides pinning and synchronization.

ATM will set the header for each cell and send the cells. When a cell arrives it will be immediately delivered to the adaptation layer. If the cell is a "last data cell," then the adaptation layer will be notified of this. The ATM layer may also notify the adaptation layer that a circuit has been disconnected. Once a circuit is disconnected, it is gone. In the event of a node or link failure, it is the responsibility of the ATM layer to attempt to reconnect. If reconnection is successful, it will be transparent to the adaptation layer.

3.1 Downcalls from an Adaptation Layer to the ATM Layer

- `atm_init()`

  This call is made from `aal7_init()`. The four daemon threads in the atm layer and the timer thread are created, and the global data structures are initialized. The network configuration file is read and processed. The routing protocol begins to work and the forwarding of arriving cells by this switch is enabled. As viewed from the aal7 layer, after this call returns, the underlying atm switch is working.

- `atm_terminate()`

  This call is made from `all7_terminate()`. The threads and their respective queues are destroyed and all allocated heap memory is freed.

- `connect(protocol_t protocol, atm_host_t node, service_t service, VCI_t &vc, unsigned int &error_code)`

  This call is made when an adaptation layer wishes to connect to the same layer on another node. The identity of the adaptation layer is provided in `protocol` (`PROTOCOL_AAL7` for example), `node` is the network address of the destination node (of course this is the the ATM node number, not the `IP` address), `service` is the service number to which a connection is being requested; this is similar to a well known port number in `TCP/IP or UDP/IP`. If the call is successful it returns ATM_CALL_OK and the reference variable `vc` returns the circuit number that ATM has allocated to identify this connection. Note that at present this is simply the virtual circuit number, but this is subject to change and the Adaptation layer should assume nothing about the meaning of `vc` other than that ATM will be able to identify the circuit given this number. If the call fails, it may return `ATM_CALL_BAD_NODE_NUM` if the destination node doesn't exist, `ATM_CALL_UNREACHABLE` if the current routing table shows a partitioned network preventing the connection, `ATM_CALL_BAD_PROTOCOL` if the `protocol` provided is not supported, `ATM_CALL_VCI_FULL` if the connection cannot be made because a VCI table along the route is full, `ATM_CALL_TIME_OUT` if a reply to the connect request did not return in a timely fashion, and `ATM_CALL_REFUSED` if the adaptation layer at the destination refused the connect request. If `ATM_CALL_REFUSED` is returned, then the reference variable `error_code` may contain a reason from the server node. This is adaptation layer specific; for example, `AAL7` will return codes indicating that there is no such service being listened for, or that the listen queue is full. This is a public member function of the `atm_c` class, and thus should be accessed through the global `atm` object.

- `disconnect(VCI_t v, unsigned int error_code)`

  This call tells the ATM layer to terminate a connection. ATM will then check to see if the circuit referenced exists. If it does, it will be torn down and `ATM_CALL_OK` will be returned. If the circuit does not exist, `ATM_CALL_FAIL`. Of course, in either case the adaptation layer should not reference this circuit again. The `error_code` argument allows the upper layer to send a reason to the other layer. This is a public member function of the `atm_c` class, and thus should be accessed through the global `atm` object.

- `send_cell_list(VCI_t v, atm_cell_list_c *list)`

  This call tells ATM to set the headers of the cells in the list, and send them out. The last cell in the list will be marked by ATM as a last cell (`ATM_PT_DATA_LAST_CELL`). The list of cells will be enqueued with the writer thread and the call will return. When ATM is done with the cell list, it will call the class member function `list->unpin()`. If this returns a new pin count of 0, the cells will be returned to the free cell list. If the upper layer doesn't want them deleted, or otherwise wants to continue to use the object (perhaps to retransmit it), then the derived class pin_synch_atm_cell_list_c, which provides synchronization and functioning unpinning, should be used instead. If this is successful it returns `ATM_CALL_OK`. If the circuit referenced does not exist, this returns `ATM_CALL_BAD_CIRCUIT`. If the cells cannot be successfully enqueued (which SHOULD NOT HAPPEN!), this returns `ATM_CALL_QUEUE_ERROR`.

3.2 Upcalls from the ATM Layer to an Adaptation Layer

Each of these calls are defined in `adaptation.h` as pure virtual member functions of `adapation_c`. Thus any adaptation layer must be a derived class of `adaptation_c` which provides functions that replace these, to provide the actual functionality.

Each of these calls is made by a daemon thread in the ATM layer and thus MUST complete in a timely fashion.

- `upcall_connect_request(service_t s, VCI_t v, unsigned int &error_code)`

  This call informs the higher layer that a request for service has come in, and that a virtual circuit has been allocated for it. The parameter `v` is the circuit number which will identify all further communication on this circuit. If the adaptation layer wishes to accept the request, it should return `ADAPTATION_UPCALL_OK`. If it wishes to refuse the connection, it may optionally place a reason in `error_code` and then return `ADAPTATION_UPCALL_FAIL`

- `upcall_disconnect(VCI_t v, atm_call_results_e reason, unsigned int error_code)`

  This call notifies the adaptation layer that the circuit identified by `v` has been terminated. A reason is passed up in `reason`. This may be `ATM_CALL_DISCONNECTED` if the adapatation layer on the other end has disconnected, in which case an `error_code` may have been sent to explain why. Other reasons are `ATM_CALL_UNREACHABLE`, `ATM_CALL_TIME_OUT`, or `ATM_CALL_VCI_FULL` which mean that the connection was broken due to node or link failure and that the reconnect procedure failed.

- `upcall_deliver_cell(VCI_t v, atm_cell_c *a, bool_e last)`

  This delivers an ATM cell to the adapation layer, which must decide what to do with it. The parameter `last` indicates whether this is the last cell in a packet. This should return `ATM_UPCALL_OK` if the adapatation layer knows about this circuit (even if it decides to drop the cell). It should return `ATM_UPCALL_FAIL`, if the adaptation layer has no record of the circuit (this would probably indicate a BIG programming error).

4 ATM

4.1 General Design

The ATM layer is handled by four primary threads. The Writer and Reader threads are very simple; the Reader thread reads cells from the UDP port which is simulating the data link layer and either forwarding them by modifying the ATM header and then placing them in the Writer thread queue, or placing them in either the Router threads queue or the ATM-VC threads queue, or placing the cell in a queue for an local port (i.e. an AAL7 SAP).

4.2 Data Structures

4.2.1 ATM cells

We implement an ATM cell class. The main data of this object will be a character array of 53 bytes. Member functions of this class will set or extract various fields from the header, as well as extract a pointer to the 48 byte atm body.

```
class atm_cell_c
{
private:
  char sCell[ATM_CELL_SIZE];

protected:
  atm_host_t sender;
  atm_host_t receiver;

public:
  atm_cell_c();
  ~atm_cell_c();

  atm_payload_t get_PT();
  void set_PT(atm_payload_t p);

  VCI_t get_VCI();
  void set_VCI(VCI_t v);

  atm_host_t get_sender(){return sender;};
  void set_receiver(atm_host_t r){receiver = r;};

  void set_data(char* cData);
  char* sData(){return (sCell + ATM_HEADER_SIZE);}


  // the only one of these used outside of the
  // DLL layer is send(atm_host_t node);
  atm_cell_status_e recv(atm_host_t &node);
  atm_cell_status_e recv(void);
  atm_cell_status_e send(atm_host_t node);
  atm_cell_status_e send(void);

  void print_cell_contents();
};
```

The PT field will determine the type of the packet. If PT is `ATM_PT_DATA_CELL` (= 0), the cell is a data cell that is not the last cell in a packet. If PT is `ATM_PT_DATA_LAST_CELL` (= 1), the cell is a data cell that is the last cell in the packet. If PT is `ATM_PT_ROUTING` (= 7), the cell is a routing protocol cell, and if PT is `ATM_PT_VC` (= 6) the cell is a virtual circuit management cell. Routing service threads will have VCI 0, while VC management cells will have either the VCI that they are directed to, or VCI 0 in the case of connect requests, for which a VCI has not yet been allocated.

For service cells we define a derived class service_atm_cell_c, which adds member functions to compute and check a checksum over the first 50 bytes of the cell. The checksum itself is placed into the 51st and 52nd bytes and the last byte is padding.

```
class service_atm_cell_c : public atm_cell_c{
public:
  compute_checksum();
  bool_e check_checksum();
}
```

The VC table manager thread and the Routing thread each derive a class from this object `vc_atm_cell_c` and `rt_atm_cell_c`, inheriting the checksum methods. These derived classes are detailed in following sections.

4.2.2 Virtual Circuit Table

For each virtual circuit we will have an entry which will contain the information pertinent to the circuit.

Virtual circuit numbers will be allocated according to the following scheme. For those circuits which are "through" circuits, i.e. circuits only passing through the node, the leading bits of the VCI will be 01. For those circuits that originate at the node the leading bits will be 11, and for those that terminate at the node, the leading bits will be 10. Since the VCI space (16 bits) is large relative to the expected maximum number of circuits on any one node (100), this does not limit us, and it provides a quick way to partition incoming cells.

To take advantage of this we maintain the VC table in three parts: Circuits starting here (at this node), circuits ending here, and circuits passing through.

For each circuit we maintain a VC table entry derived from the following class:

```
class VC_table_base_entry_c
{
protected:
  // The Atm Address of the node initiating the connection.
  atm_host_t Initiator, Destination;

  // The serial Number that makes this unique
  serial_t InitiatorSerialNum;

  // The upper level that this VC services (will always be AAL7)
  protocol_t Protocol;

  // The state this connection is in
  atm_state_e State;
```

```
  // This entries VCI
  VCI_t this_vci;

public:
  // There are many public accessor functions, but including
  // them would not be helpful
};
```

Each circuit will be uniquely identified by the Initiator, and the InitiatorSerialNum (a 64 bit integer). This allows us to be certain when matching existing VCI entries with VC message cells from other nodes. The rest is useful to handle the connection.

For through connections, we maintain entries that are defined as follows:

```
class VC_table_through_entry_c : public VC_table_base_entry_c
{
private:
  atm_host_t PrevHop, NextHop;
  VCI_t PrevVCI, NextVCI;

  // leaving out accessor functions
};
```

This allows us to take a cell coming in, check the address it came from against the hops, and direct it to the correct next destination, changing the VCI accordingly.

Finally for endpoint connections we have

```
class VC_table_endpoint_entry_c : public VC_table_base_entry_c{
private:
  atm_host_t Hop;
  VCI_t VCI;

  // wait on the condition variable for outgoing connections.
  cond_t *cond;         // protected by the VCT lock
  int atm_result;      // result of the connect request.
  unsigned int reply;  // if the result was an error, the reply from the
                       // other side

  // leaving out accessors again...
```

For a VCI at an endpoint, we know whether it is incoming from another node or coming from. Thus we know which direction to route it. The condition variable and result and reply, are used to wait during connection establishment at an initial node. (This is the only place in the ATM layer where a thread blocks on anything other than a synchronized queue. It just seemed easier....)

Finally all of these are combined into one big virtual circuit table, (for clarity we omit the many private methods)

```
class VC_table_c{
  VC_table_c();
  ~VC_table_c();
```

```
  // Forwards a data cell according to it's VCI
  // and where it came from.
  // TODO :
  void forward_cell(atm_cell_c *c);

  // Functions to handle a virtual circuit management message.
  // This is the KEY entry point that every message must pass through.
  void handle_message(VC_atm_cell_c *message);

  // This puts the message onto the queue
  void deliver_message(VC_atm_cell_c *message);

  // handle requests from above
  atm_call_results_e connect(protocol_t protocol, atm_host_t node,
       service_t service, VCI_t &vc,
       unsigned int &error_code);

  atm_call_results_e disconnect(VCI_t v);

  atm_call_results_e send_cell_list(VCI_t v, atm_cell_list_c *a);

  void manage();

  void print_table(FILE *stream);

private:
  // we actually DO have a table hidden in this class.  Actually
  // three tables:
  VC_table_through_entry_c* through_table[VC_MAX_NUMBER_CIRCUITS];
  VC_table_endpoint_entry_c* init_table[VC_MAX_NUMBER_CIRCUITS];
  VC_table_endpoint_entry_c* dest_table[VC_MAX_NUMBER_CIRCUITS];

  // bookkeeping...
  int iLastThrough, iLastInit, iLastDest;
  int iNumThrough, iNumInit, iNumDest;

  // now the good stuff...

  mutex_t lock;
  serial_number_c serial_here;

  // the event queue for the manager...
  // events are neighboring node down, timeouts, cell delivered
  sync_queue_c<event_c *> messages;
```

When an incoming data cell arrives, it will be passed to `forward_cell`, which will either modify and send it out, or deliver it up to an adapation layer, as appropriate.

When an incoming message arrives, it will be passed to `deliver_message`, which will place it in the event queue for the manager. The manager will extract it and pass it to `handle_message`. (Actually always passing pointers, not cells of course).

When a request comes down from above to either connect, disconnect or send a list of cells, the upper layer thread will pass through `atm_c`, and to the three matching methods here.

This is currently protected by a single `mutex_t`, but may be changed to multiple locks (one for each of the three sub-tables) or a reader-writer lock depending upon performance benchmarks.

4.3 Virtual Circuit Protocol

4.3.1 Overview

The virtual circuit protocol is monitored by the VC Table manager thread. This thread receives events which direct its control of the operation of all virtual circuits on the node. It will wait on its event queue, and whenever an event arrives it will be woken up. Most events will simply be management cells arriving from neighboring cells, but there will also be timeout events from the timer thread and node-down events from the router thread. (Note: There are no node-up events because that information is conveyed through the routing table. Node down is also conveyed by the routing table, but the consequence of ignoring it is only keeping inefficient established circuits, as opposed to not rebuilding broken circuits.)

4.3.2 Virtual circuit control message types

The format of the 53 byte virtual circuit message cell is detailed in Figure 3. Note that we have purposely aligned the 4 byte data integers so that they will be aligned in memory space when compiled with the g++ compiler on Solaris, to make copying into and out of the data fields faster.

The fields in Figure 3 are:

Signal Type This is the type of VC signal that is being sent. Possible values are *VC_CONNECT*, *VC_CONNECT_ACCEPT*, *VC_CONNECT_REJECT*, *VC_TEARDOWN*, *VC_TEARDOWN_REPLY*.

Subsignal Type This contains a submessage that explains exactly why the signal was sent. Possible subsignal values for the signal *VC_CONNECT* are *VC_SUB_NORMAL* and *VC_SUB_RECONNECT* which indicate whether this is a normal connection or a reconnection request; this only matters at the destination. *VC_CONNECT_ACCEPT* can only have subsignal *VC_SUB_NORMAL*. *VC_CONNECT_REJECT* can have sub signals *VC_SUB_REJECTED* which indicates that the higher layer at the destination has rejected the request, *VC_SUB_TABLE_FULL* which indicates that some node along the way has a full table, and *VC_SUB_UNREACHABLE* which indicates a network connectivity problem that arose during the connection procedure. *VC_TEARDOWN* can have sub signals *VC_SUB_NORMAL* which indicates a teardown by the adaptation layer at the other end of the connection, and *VC_SUB_RECONNECT*, which indicates that network connectivity has failed, breaking the connection, and that reconnection should be tried; this matters only to the two enpoints of the connection. Finally, *VC_TEARDOWN_REPLY* can have sub-signals *VC_SUB_NORMAL* or *VC_SUB_BAD_VCI*. These differ only in helping programmers detect errors.

Initiator Address This is the ATM network address of the initiator of the circuit. This field is a 64 bit field that is equivalent to a 64 bit integer. This is obviously a larger address space

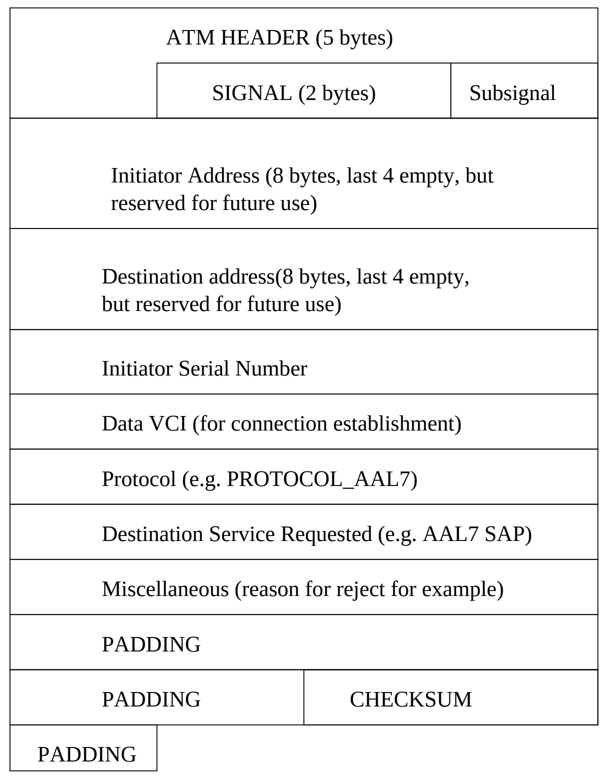

Figure 3: Layout of data in vc_atm_cell_c.

than is required at present, but will provide adequate room for many addresses if our CS640 project takes over the world. At present we use 32 bit addresses which are placed in the first 32 bits of the 64 bit field. The other 32 bits are unused at this time.

Destination Address This is the address of the destination node. See the discussion of the Initiator Address, above for more details.

Initiator Serial Number At each ATM node, a serial number counter is maintained. At the node initiating a connection, the first connect request will be stamped with this number. This number will be stored at every node that is part of the virtual circuit. This and the two above address fields will be sent with all virtual circuit control messages associated with a particular circuit and this will enable us to ensure that messages are applied to the correct circuit.

VCI In *VC_CONNECT* and *VC_CONNECT_ACCEPT* messages this contains the VCI at the node sending the message. This is so that each node will know which VCI to send to for messages moving forward or backward over the circuit. In other messages, it is undefined.

Protocol This is the adaptation layer that is associated with this virtual circuit. This will be a numeric constant. The only valid destination layers in this project will be *PROTOCOL_AAL7* and any simple test layers we choose to define (*PROTOCOL_AAL_SIMPLE*) This only used in *VC_CONNECT* messages.

Destination Service This is the service number of the protocol to attempt to connect to (e.g. the SAP in AAL7). It is only used in *VC_CONNECT* messages.

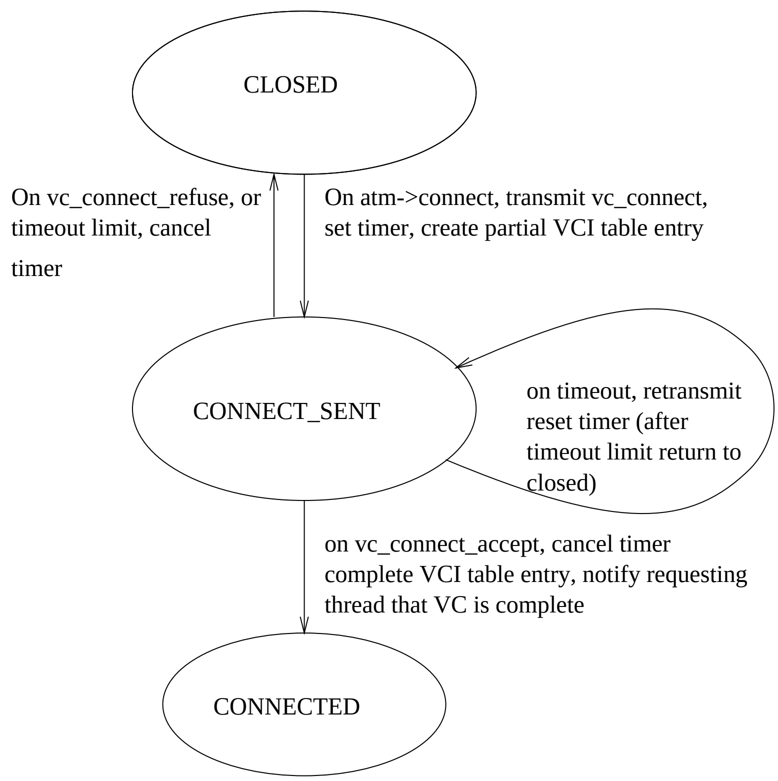

Figure 4: Connect at initial node.

Miscellaneous This will be used for any miscellaneous information that needs to be conveyed. In particular, when an adaptation layer disconnects or refuses a connection, it can send back an unsigned integer reason. The ATM layer carries that message here.

Checksum This will be calculated with the standard Internet checksum algorithm. This is further described with the service_atm_cell_c class above.

4.3.3 Protocols on individual nodes

Establishment of virtual channel The overall idea is that each node picks its VCI and sends it to the next hop (as determined from the routing table) in a connect message and sends the VCI back to the previous node in the *VC_CONNECT_ACCEPT* message.

Establishment at source node The state diagram for this action is located in Figure 4.

At the node which is establishing a virtual circuit connection, the calling thread (presumably a higher layer such as *AAL7*, or perhaps a user thread that is directly interfacing with *AAL7*) will initiate the process. It will call `atm->connect` which will sanity check the request, form a message with *VC_CONNECT*, create a partially filled VC table entry which will be in the CONNECT_SENT state, and then the thread will schedule a timeout for the VC table manager thread (on this VCI) and then wait on the VC table entry's condition variable for a reply.

When a *VC_CONNECT_ACCEPT* or *VC_CONNECT_REJECT* arrives, the VC table manager thread will move the connection to the CONNECTED or CLOSED states, and then wake up the

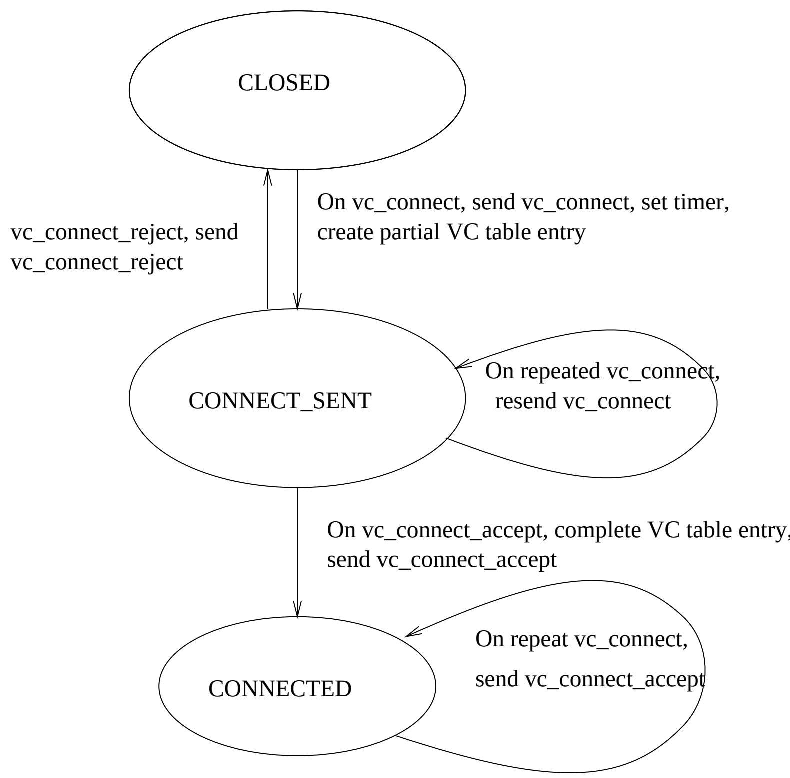

Figure 5: Connect at an intermediate node.

calling thread. The calling thread will check the state, delete the VC table entry if the state is CLOSED, and return to the higher layer.

If no reply arrives, the timeout will occur and a retransmit will be attempted (up to a limit of 3 attempts).

Establishment on an intermediary node The state diagram for this action is located in Figure 5.

At an intermediary node, the VC table manager thread receives the connect message. It checks existing connections to see if the message is a repeated request (based on serial number and address of initiator). If the request is new, the node checks the routing table to find the next hop. It picks a new VCI for the new circuit (if one is available), enters a partially completed entry into the virtual circuit table. If there is no problem with the routing table or the VC table, it sends a connect message to the next hop, moving from CLOSED to CONNECT_SENT (of course if there is a problem, a *VC_CONNECT_REJECT* message is sent back, and all traces of the circuit eliminated).

If a *VC_CONNECT_ACCEPT* reply is returned, the VCI from the next hop is entered into the partial entry in the virtual channel table. Then the VCI for this channel is entered into the *VC_CONNECT_ACCEPT* message and sent to the previous node, and the connection on this node is in the CONNECTED state. If a *VC_CONNECT_REJECT* message was received, the partial entry in the virtual channel table is removed and the *VC_CONNECT_REJECT* message is sent back to the previous node and the state returns to CLOSED.

If the request is a duplicate, and the node is in CONNECT_SENT state, a repeat *VC_CONNECT*

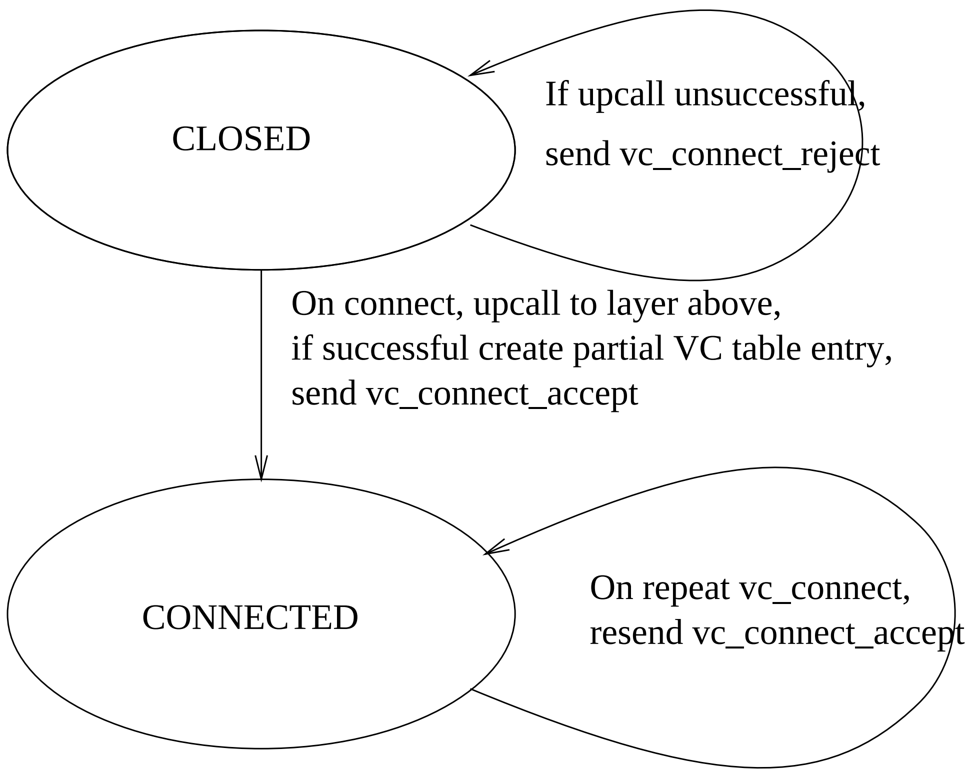

Figure 6: Connect at destination node.

is sent forward (this will happen if the initial node times out). If the request is a duplicate and the node is in CONNECTED state, a *VC_CONNECT_ACCEPT* message is sent back. Note that if the state is CLOSED, all information about the attempted connection will have been erased, so the node views the message as a new request.

Note that at an intermediate node, there is no difference between a *VC_CONNECT* with sub-message either *VC_SUB_NORMAL* or *VC_SUB_RECONNECT*, so we can handle these with one function (just make sure to forward the sub-message as well as the message).

Establishment at destination node The state diagram for this action is located in Figure 6.

When a request arrives at the destination node, the ATM VC table manager thread picks a VCI for the entry, and makes an upcall to the appropriate higher layer to see if the higher layer is listening on the requested port. Two replies can occur. The first is that either there is no listener on this port or the listen queue is full. If this occurs, the thread stays in CLOSED state and sends a *VC_CONNECT_REJECT* message back. If the request goes onto the listen queue or is immediately accepted the adaptation layer will accept the connection and the circuit enters CONNECTED mode. The VC table manager finishes the VC table entry and sends a *VC_CONNECT_ACCEPT* reply to the previous node, and enters CONNECTED mode. If a repeat connect message is received, the *VC_CONNECT_ACCEPT* is re-sent.

Tearing down virtual circuits The overall plan is that each node will receive a *VC_TEARDOWN* and send a *VC_TEARDOWN_REPLY* in one direction, send a *VC_TEARDOWN* and receive a

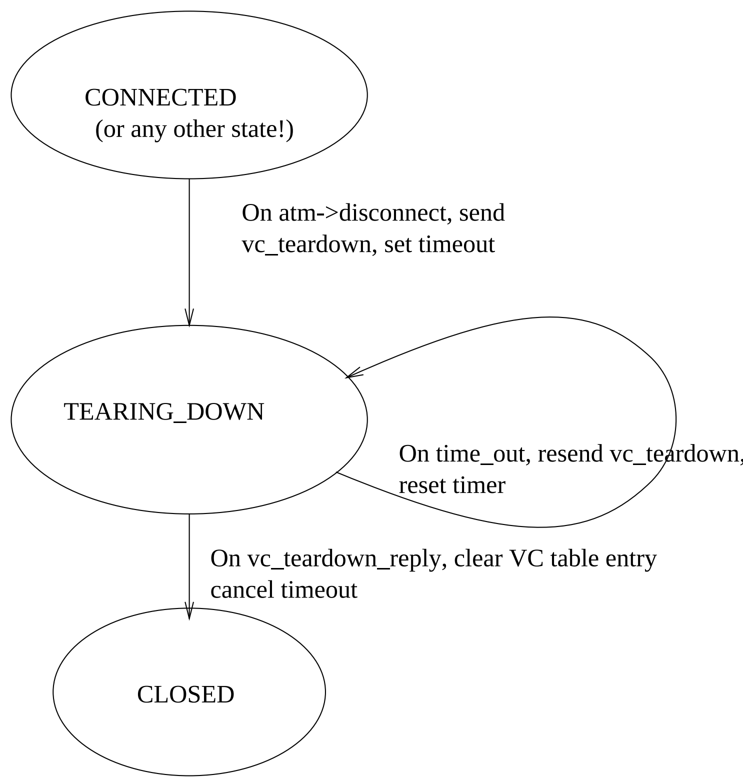

Figure 7: Tearing down at endpoint node initiating teardown.

VC_TEARDOWN_REPLY in the other direction. Note that at the endpoints *VC_TEARDOWN* with submessage *VC_SUB_RECONNECT* are treated specially as detailed below, while, at an intermediate node, the submessage matters only inasmuch as it must be copied into any corresponding messages. The discussion here is for the case of actual tearing down of a circuit. The case in which the teardown is caused by a downed node or link is handled below in **NODE down and reestablishment**. Thus this section only applies to *VC_TEARDOWN* messages where the submessage is *VC_SUB_NORMAL*.

Also note that in all state diagrams, we have omitted (for clarity) the cases in which a node in a state other than connected receives a *VC_TEARDOWN* message. These should be handled essentially the same as if the state was CONNECTED.

Further note if a *VC_TEARDOWN_REPLY* is lost, the circuit will be closed, but a repeat *VC_TEARDOWN* will be sent. Thus a *VC_TEARDOWN* on a nonexistant circuit will reult in a *VC_TEARDOWN_REPLY*.

Tearing down at a closing endpoint The state diagram for this action is located in Figure 7.

At the endpoint that is initiating the close, the VCI table entry is cleared, a *VC_TEARDOWN* message is sent and a timeout is scheduled. If a *VC_TEARDOWN_REPLY* is received the timeout is cancelled, and the connection is in the CLOSED state. If the timeout occurs the *VC_TEARDOWN* is re-sent and the timeout rescheduled (no limit since eventually we will get a reply, or we will find the node is down).

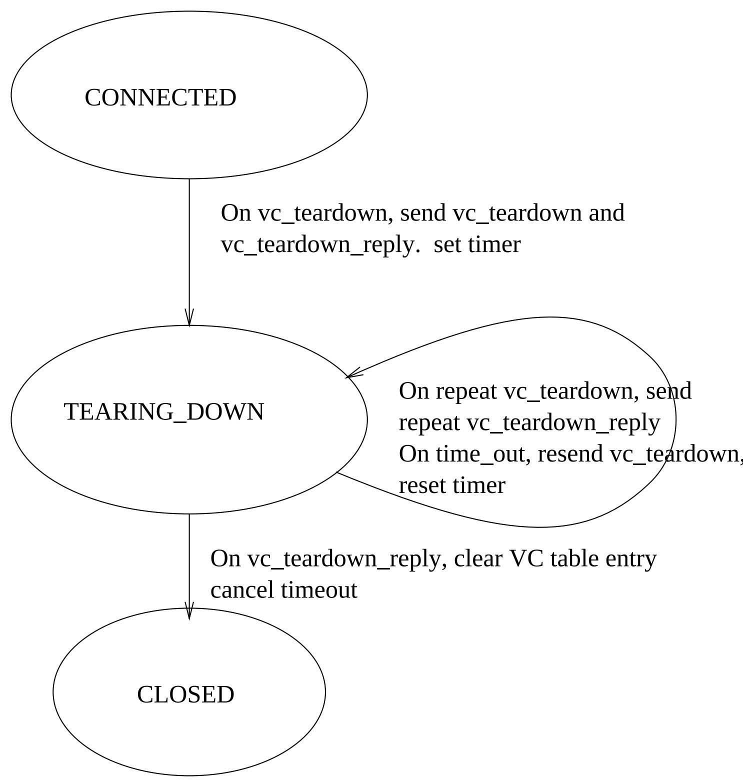

Figure 8: Tearing down at intermediate node.

Closing at intermediary node The state diagram for this action is located in Figure 8.

When an intermediary node receives a *VC_TEARDOWN* message, it checks the virtual circuit table to confirm that the circuit being torn down exists (it is possible that it won't if this circuit has already been torn down, but the reply was lost.) If the circuit doesn't exist, a *VC_TEARDOWN_REPLY* is sent back. If the circuit does exist, the connection will enter the TEARING_DOWN state and will send a *VC_TEARDOWN* further along the circuit, and send a *VC_TEARDOWN_REPLY* to the node that sent the teardown. The connection will wait in the TEARING_DOWN state for a *VC_TEARDOWN_REPLY* to its *VC_TEARDOWN* message, resending if the message times out.

If a repeated *VC_TEARDOWN* is received while the connection is in TEARING_DOWN, the *VC_TEARDOWN_REPLY* is re-sent. If a new *VC_TEARDOWN* is received from the opposite direction, a reply is sent and the virtual table entry for this connection is removed.

Closing at open endpoint The state diagram for this action is located in Figure 9.

If a *VC_TEARDOWN* is received at the open endpoint, we first check the sub-signal to see whether this was sent from the other end of the connection or the connection was broken and should be reestablished. The reconnect case is discussed below. If the teardown is happening because the other end disconnected, an upcall is made to the higher layer to terminate the mapping from VCI to protocol port. A *VC_TEARDOWN_REPLY* is sent and the virtual circuit table entry is removed from the table.

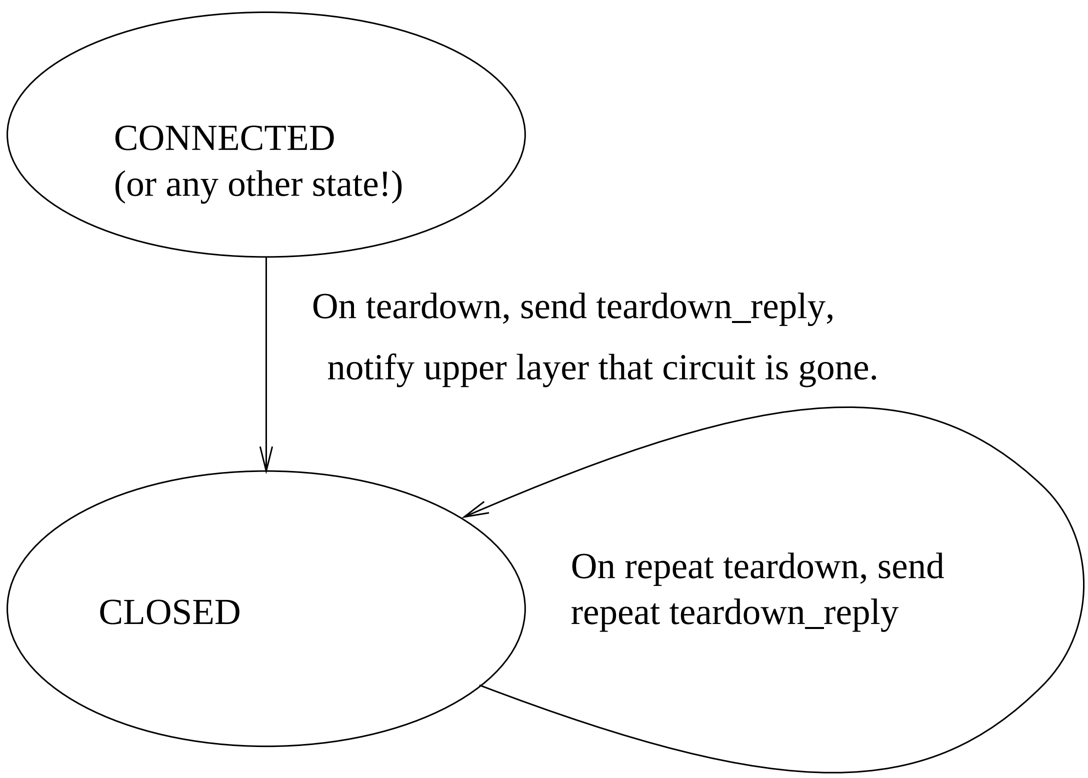

Figure 9: Tearing down at endpoint node initiating teardown.

Node down and reestablishment The basic stategy is for nodes at either end of a down link to send a *broken* message toward the end of the chain each can reach. These messages are *VC_TEARDOWN* with submessage *VC_SUB_RECONNECT* which functions precisely like *VC_TEARDOWN* with *VC_SUB_NORMAL* messages except at the endpoints. The destination endpoint will set a timeout, expecting a reconnect message to arrive. The initiating endpoint will wait a short time for the routing table to stabilize (the routing thread will have initiated the routing process before notifying the virtual channel thread that a link has been broken.) After waiting, it will attempt to send a reconnect message toward the destination. Only if no new path exists will the higher level be notified that the connection has been broken. If a node goes down, then each of the bracketing nodes will detect the down links, so focusing only on down links rather than down nodes is more general.

If an endpoint of a circuit goes down, either reconnecting will not be initiated at the initiator, causing the destination to give up on a reconnect, or the reconnect will not be sent and thus be unable to reach the destination and the destination will timeout and give up.

Node down next to intermediary node This describes how a node that discovers a neighboring node has gone down handles any virtual circuits passing through that node.

The state diagram for this action is located in Figure 10.

When the routing thread detects that a connection to a node is down (this of course includes the node at the other end of the link being down), the ATM VC table manager thread is informed. At this point the manager thread scans the VCI table. For any virtual circuit that connects (either ahead or behind) over that link, the failure has to be dealt with. For each virtual circuit in the

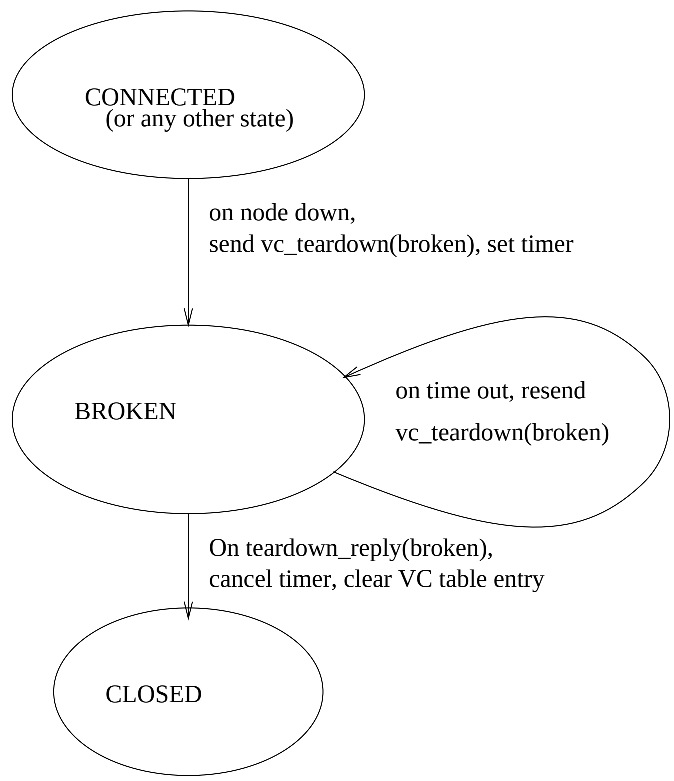

Figure 10: Node down - actions at midpoint.

CONNECT state, the thread moves to the BROKEN state and a *VC_TEARDOWN* message with a sub message *VC_SUB_RECONNECT* is sent to the node in the opposite direction from the down link. Once a *VC_TEARDOWN_REPLY* is received the entry for this virtual circuit is removed from the VC table. In the pathological case that links have gone down in both directions, nothing can be done but remove the entry in the VC table.

Protocol at intermediate nodes As far as intermediate nodes are concerned, all *VC_TEARDOWN* messages are the same and handled as above. The only detail is to be sure to forward the sub-message field, so that the endpoints will do the correct thing.

If messages attempting to reconnect pass through an intermediate node, the intermediate node will function exactly as in the connection protocol above, since intermediate nodes will have no memory of the connection.

Protocol at initiator node The state diagram for this action is located in Figure 11.

When the node that initiated a connection receives a *VC_TEARDOWN* with sub-message *VC_SUB_RECONNECT* it sends a *VC_TEARDOWN_REPLY*, enters the BROKEN_INITIATOR state and pauses to allow the routing table to stabilize (the node that detected the down link would have initiated the routing protocol). As always, if it receives a repeat of the *VC_TEARDOWN* message it resends the *VC_TEARDOWN_REPLY*.

Once the time to allow the routing table to stabilize has passed, the connection sends a *VC_CONNECT* message with sub message field *VC_SUB_RECONNECT*. The connection then enters the RECONNECT_SENT state. At this end, this directly parallels the CONNECT_SENT

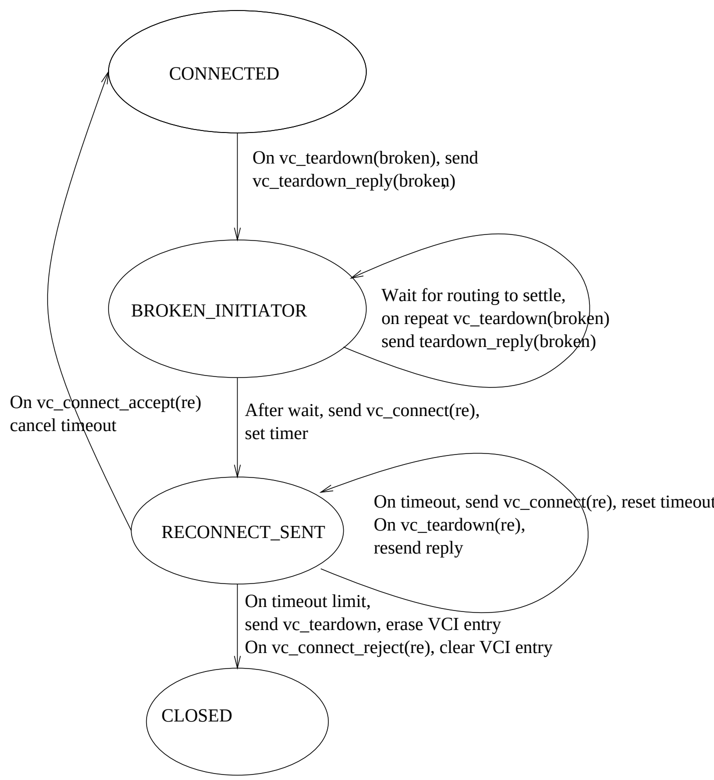

Figure 11: Node down - actions at initial node.

state and the connect protocol.

Protocol at destination node The state diagram for this action is located in Figure 12.

When a destination node of a connection receives a *VC_TEARDOWN*, it sends a *VC_TEARDOWN_REPLY*, enters the BROKEN_DESTINATION state and then pauses waiting for a reconnect message to arrive. When a reconnect message arrives, the connection returns to the CONNECTED state; a *VC_CONNECT_REPLY* message is sent and from then on the circuit is handled as in **Establishment at destination node** above.

4.4 Routing

The VC signalling protocol has to refer to the routing table in order to establish a VC between two nodes. The routing protocol is responsible for building and maintaining the routing table based on the connectivity status of adjacent nodes and based on information sent to it by other nodes.

4.4.1 Routing Thread

There is only one thread associated with the routing protocol. The thread blocks on its event queue waiting for either timer events or service cell arrival events. Timer events are generated when it is time to send echo cells to the neighboring nodes, when it is time to resend connectivity information cells to the neighboring nodes because they were not acknowledged, and when it is time to declare one of the neighboring nodes down because an echo cell from that node has not arrived. Service cell arrival events are generated when a special cell carrying routing information arrived.

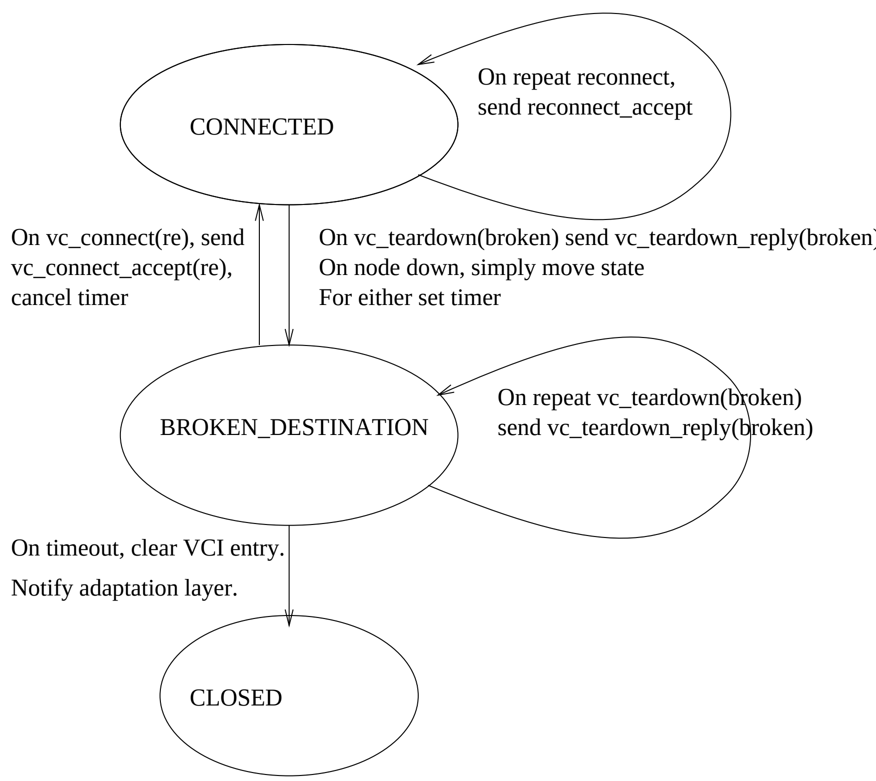

Figure 12: Node down - actions at destination node.

See Figure 13 for the simple "finite state machine" for this protocol. Note that the only "states" here are "CLOSE" which is used at startup and shutdown, before the Routing thread handler is started by `atm_c::route`. The other "state" is "WAIT" which indicates that this thread reacts to events (including timing events which include lack of "pong" events).

4.4.2 ATM Routing Service Cells

Service ATM cells in the routing protocol are used for reliable flooding of link-state information and for detection of link failures. A routing service ATM cell has the following fields: message (service) type, virtual node number of the sender, virtual node number of a neighboring node, status, serial number, and checksum (Figure 14).

There are three types of routing ATM service cells:

- Link-State Cell

  This cell carries connectivity information. For example, if node 4 wanted to notify other nodes in the network that one if its neighbors, node 7 is down, it would send the following information: message type = RT_LINK_STATE (0), atm node = 4, neighbor node = 7, status = RT_DOWN (0).

  To assure reliability of the routing protocol, this cell has to be acknowledged by the recipient. The sending node sets up a timer event for 300ms. If the timer expires before an acknowledgment is received, the sender retransmits the original cell again.

- Link-State Ack Cell

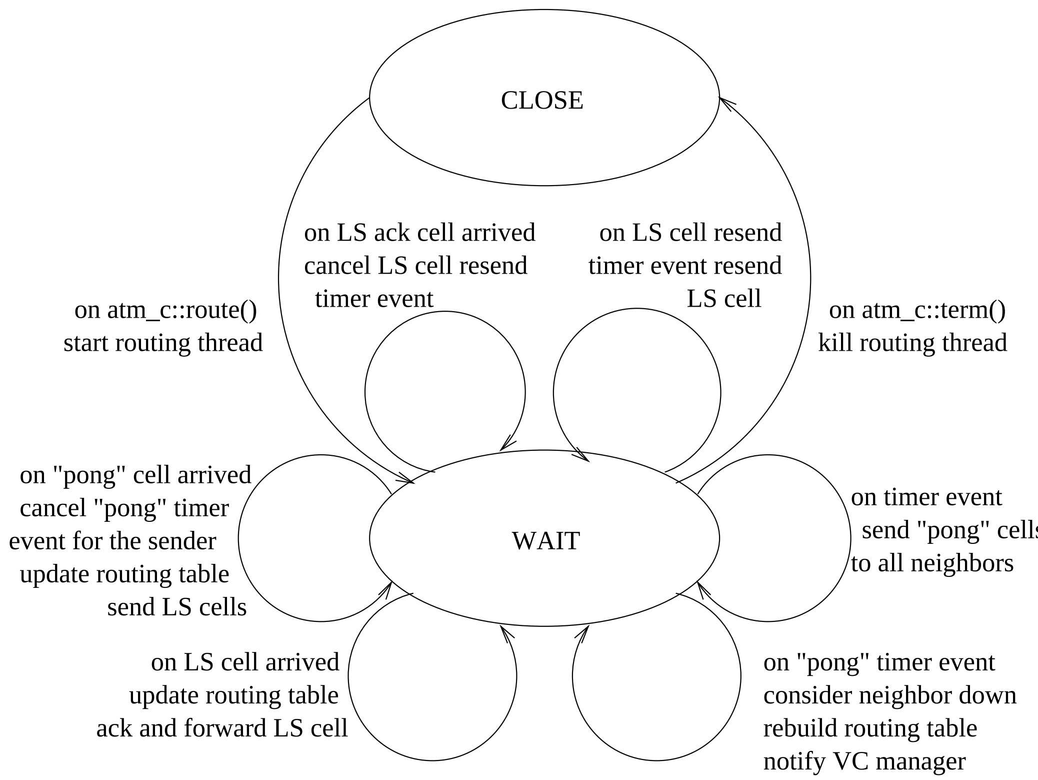

Figure 13: Finite state machine for routing protocol (Only one state).

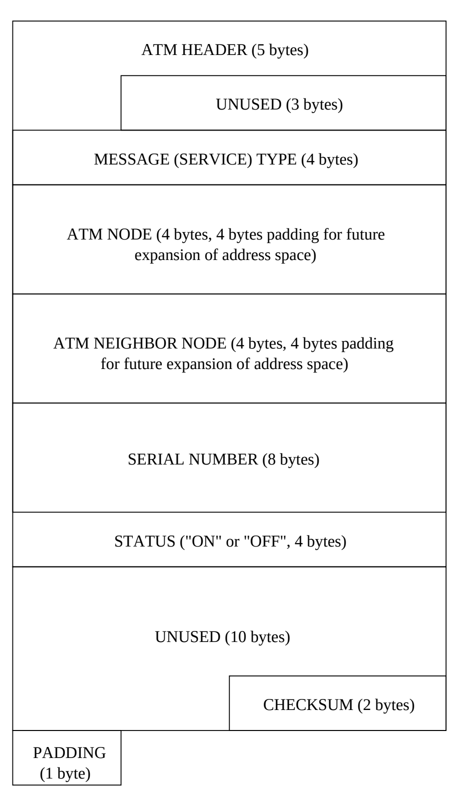

Figure 14: Layout of data in ATM Routing Service Cells.

This cell acknowledges a receipt of a link-state cell. Message type field is set to RT_LINK_STATE_ACK (1) and the rest of the fields are left unchanged. When the sender of the original link-state cell receives its acknowledged version, it compares their serial numbers. If they match, the sender assumes that the original cell was received and removes a timer event it scheduled for a retransmission of the cell.

- Echo "pong" Cell

  This cell is used by each node in the network to let its neighbors know that it is up. The only field used in this cell is message type which is set to RT_PONG (2). These cells are sent every 100ms and are not acknowledged. If a node does not receive one of these cells from one of its neighbors for 300ms, it assumes that the neighbor node is down.

4.4.3 Routing Structures

The routing table is a two dimensional array of the following structures indexed by virtual node number.

```
typedef struct rt_entry_s {
  atm_host_t next_hop;   // next hop node
  unsigned cost;          // cost to get to destination node
  unsigned serial_number; // detects old information
} rt_entry_t;
```

Given the current node and the destination node, we can obtain the next hop node or, if cost is set to infinity (MAX_NODE + 1), we know that there is no such node in the network or that it is no longer reachable from the current node. The routing information is sent in a special service cell. There are two types of routing service cells: link-state cell and "pong" cell. Link-state cells carry the following information: message (service) type (RT_LINK_STATE), virtual node number of the sender, virtual node number of a neighboring node, the status of the link ("up" or "down"), and a serial number used to stop cell from further flooding and also used to acknowledge a cell. These cells are used to describe the connectivity status of a node. The "pong" cells are sent by each node to its neighbors letting them know that a node is up and carry service type (RT_PONG) information only since the cell is not forwarded and DLL knows where the cell came from. The payload type of a service cell is set to ATM_PT_ROUTING. Because the routing table is used by two threads: VC manager thread and routing thread, it is protected by a mutex_t. As a possible improvement we will change this to a reader writer object.

4.4.4 Routing Algorithm

We use a slightly modified link state algorithm to build and maintain the routing table. The main differences are (i) we segment the imaginary link state packets into several ATM cells and (ii) the events that trigger a flooding of these cells are different.

The routing table is first built when the node starts. The DLL module parses the network configuration file and, while building the `vn_to_ip_map`, calls `add_link( atm_host_t N, atm_host_t L)` method of the `atm_rt_c` class that simply registers the fact that the node `N` has a direct link to node `L`. Once the file is parsed the `build_table` method, implementing the Dijkstra's shortest-path algorithm, is called. It rebuilds only the part of the routing table associated with the current node. This allows us to use this structure as both the routing table for the current node and as a repository of topological information about the entire network which can be used to rebuild the table, and reroute all open VCs, if one of the adjacent nodes goes down. The repository is dynamic in a sense that its entries can be changed based on routing information sent by other nodes.

```
void
atm_rt_c::build_table(atm_host_t vn)
{
  // temporary array of nodes that have been
  // found to be reachable by some shortest path
  unsigned temp[MAX_NODE + 1];

  // right now no node is found to be reachable
  for (int t = 1; t <= MAX_NODE; t++) {
    temp[t] = 0;
  }

  for (atm_host_t n = 1; n <= MAX_NODE; n++) {
    if (n != vn) {
      // assume node is not reachable
      unsigned min = MAX_NODE + 1;        //distance
      unsigned min_node = MAX_NODE + 1;
      // find the node reachable by shortest path
      for (atm_host_t l = 1; l <= MAX_NODE; l++) {
        if (rt[vn][l].cost > 0 && temp[l] == 0) {
          if (rt[vn][l].cost < min) {
            min = rt[vn][l].cost;
            min_node = l;
          }
        }
      }
      // if node not found (all processed already) we are done
      if (min_node == MAX_NODE + 1) {
        break;
      }
      // mark the node as processed
      temp[min_node] = 1;
      // replace paths of this node with the paths of the
      // node we just found if those paths are shorter
      for (atm_host_t mn = 1; mn <= MAX_NODE; mn++) {
        if (min_node != mn) {
          if (rt[min_node][mn].cost + rt[vn][min_node].cost
             < rt[vn][mn].cost) {
            rt[vn][mn].cost = rt[min_node][mn].cost + rt[vn][min_node].cost;
            rt[vn][mn].next_hop = min_node;
          }
        }
      }
    }
```

```
  }
}
```

The routing protocol is responsible for detecting any changes in the topology of the network. A special cell that we call "pong" is sent several times a second (typically 1 cell every 100ms) to each neighboring node. This signals "upness" of the node to its neighbors. If such a cell is not received within 300ms, we declare the node that was supposed to send it down. When this happens, the appropriate entry in the routing table is changed to reflect the fact that this node is not reachable. The table is then rebuilt to allow for rerouting of all the paths that may be passing through that node. Only then does the routing protocol notify the VC manager that a node is down by appending an event to its event queue. Since the routing table is already rebuilt, the VC manager is able to quickly reroute any existing VCs passing through the node that went down.

The routing protocol sends another type of service cell, called link-state cell to each of the neighboring nodes which is still up to tell it that another node is down. This cell is flooded to all the nodes. The flooding is stopped when nodes stop forwarding the cell because its serial number is not higher than the one they already have.

Unlike the "pong" cell, the link-state cell has to be acknowledged by the destination node. Therefore, the sender node sets up a timer event for every link-state cell it sends. If the time expires, before an acknowledgement is received, the cell is sent again.

If a node that was down comes back up, it has to wait at least 300ms before it can start sending the "pong" cells. This guarantees that the neighboring nodes will notice that the node was down and will start tearing down and rerouting any existing VCs going through that node. When the neighbors of a node that was down see that it came back up, they send link-state cells reflecting that fact to all of their neighboring nodes. Eventually, as in the situation when a node went down, these cells will be flooded to all nodes in the network allowing them to update their routing tables.

The difference between this type of flooding and the one described above is that each node has to forward link-state cells it receives from other nodes if the serial numbers in those cells are higher than the ones they already have in their routing tables, and, if the information contained in a cell changes their routing tables, they also have to send link-state cells describing their connectivity with their neighboring nodes. This is to assure that all those cells reach the node that was down.

When a node that was down comes back up, it assumes that all links in the network are down. However, it uses its network configuration file to send the "pong" cells to its neighbors telling them that it is back. Those initial "pong" cells trigger a flooding of link-state cells from all nodes on the network which allows the node to rebuild its routing table and the repository.

4.4.5 Interface

```
class atm_rt_c {
 public:
  atm_rt_c();
  ~atm_rt_c();

  atm_rt_status_e route(void);
  atm_rt_status_e add_link(atm_host_t vn, atm_host_t l);
  void build_table(atm_host_t vn);
  atm_rt_status_e next_hop(atm_host_t dest, atm_host_t &next);
  atm_rt_status_e term(void);
...
```

```
 private:
  // routing table for the current node and repository of link-state cells
  rt_entry rt[MAX_NODE + 1][MAX_NODE + 1];
  // synchronized queue of events
  synch_queue_c<event_c*> q;
  mutex_t rt_lock;
...
};
```

The `route()` method starts the routing thread that blocks on a synchronized queue of event_c class objects. The `add_link()` and `build_table()` methods are described above. The `next_hop()` method returns the next node on the path to a destination node. The `term()` method "kills" the routing thread.

5 The AAL7 Layer

The `AAL7` layer is one example of an adaptation layer that may interface with `ATM`. It is the only example that we will detail here.

5.1 Data Structures

In this section we will describe the data structures used in the AAL7 Layer. Data structures are *shared*, if they are potentially accessed by more than one thread of control. For shared data structures, synchronization is necessary. We first describe the shared data structures that reside in the AAL7 layer.

There are three shared data structures in the AAL7 layer: the listen table, the active connection table and the type table. In the *listen table* we store information about service points, that have been established through a call to `aal7_listen()` from the application layer. An entry in the listen table consists of the following three fields:

1. a local descriptor for the SAP (the return value of `aal7_listen()`),
2. the SAP, and
3. a synchronized queue of virtual channel identifiers (we will refer to this queue in the remainder as the *listen queue*); the size of the queue is given by the argument `limit` in the call `aal7_listen()`.

The AAL7 layer does not know that the identifiers are actual virtual channel identifiers in the ATM layer; it just treats them as descriptors to communicate with the ATM layer. To avoid confusion, we will refer to them as virtual channel identifiers also in the AAL7 layer. The *active connection table* has two fields:

1. a local descriptor for the connection (the return value of `aal7_connect()` or `aal7_accept()`), and
2. a connection descriptor (the connection descriptor is explained in detail below).

The *type table* maps virtual channel identifiers to entries in either the listen table or the active connection table. The table has three fields:

1. a virtual channel identifier,
2. a type-flag (either `AAL7_IN_LISTEN_TABLE` or `AAL7_IN_CONNECTION_TABLE`), and
3. the index of the corresponding entry in the listen table or the active connection table

A connection descriptor contains the following fields:

- incoming cell queue (the *data queue*)
  All cells that arrive at the atm layer for this connection will be put into the data queue through an upcall `upcall_deliver_cell()`. Note that an event is not necessarily associated with enqueueing the cell. This mechanism allows to wake up a connection thread only if it has actually enough information to process the data (i.e., check correctness through checksum, extract trailer, etc.), namely, a whole packet has arrived.
- event queue
  All communication to a connection thread is handled through its event queue. The connection thread waits blocking on this queue. The upcall `upcall_deliver_cell()` will put a cell into the data queue and if the cell is the last cell in a packet, the upcall function will also put an event with the id `AAL7_EV_LAST_CELL` into the event queue. Also timeouts that have been scheduled by the AAL7 layer are handled through this queue. If a timeout occurs, the timer object puts an event with the id `AAL7_EV_ACK_TIMEOUT` into the event queue. All types of events in the AAL7 layer are described in detail in Section 5.3.
- virtual channel identifier
  The virtual channel identifier that is used for all communication with the ATM layer.
- unsent cells queue
  A queue of pointers to atm cells that the application layer has already handed to the connection through an `aal7_send()`, but which have not yet been sent due to the credit schema.
- unacknowledged packets queue
  A queue of pointers to AAL7 packets; these packets have been sent, but no ACK has been received yet for them.
- last ACK received
  The number of the last ACK received
- received packets list
  A list of packets that have been received but that have not been handed up to the application layer through a call `aal7_recv`.
- the receive offset
  An offset into an ATM cell that indicates up to what byte data has already been handed to the application layer. This offset is always into the first cell of the first packet.
- advertised window size on the other size
  This is the offset from the last acknowledgement received.

In addition, a connection descriptor contains also the following fields, which ease the synchronization with a blocking application layer call (like `aal7_recv()` and `aal7_disconnect()`):

- a flag showing whether there is an application thread waiting, i.e., there has been a call to `aal7_recv()` which blocked because the received packets list was empty.

- a condition variable on which the application thread is waiting

5.2 Threads

Now that we have the shared data structures of the AAL7 layer described, we can give an overview over the assignment of tasks to threads in the AAL7 layer.

There are no global service threads in the AAL7 layer. Threads exist only for established connections; we call these threads *connection threads*. There are exactly two cases in which a connection thread is created:

1. Through a call to `aal7_connect()`
   The application thread makes a call to the atm layer trying to establish a connection to the remote SAP. If this connection is successful, the application thread sets up the necessary data structures for the connection and creates a connection thread that waits on its event queue (for a packet with a SYN from the other side, see Section 5.5).

2. Through a call to `aal7_accept()`
   In a call to `aal7_accept()`, the application thread waits blocking on the listen queue of the SAP. It then takes the first entry from the listen queue, sets up the necessary data structures, and creates a thread that will handle the connection.

Note that in both cases the application thread handles the setup of the shared data structures for the connection. At this point the ATM layer has already established a virtual circuit between the two endpoints. The application thread first allocates a connection descriptor and initializes it. It then makes the respective entry in the active connection table. Only after having set up these data structures, the application thread spawns the connection thread and returns. This is due to the fact that as soon as the necessary data structures are in place, upcalls to the AAL7 layer from the ATM layer for this connection will be handled properly, since they only manipulate the data queue and/or the event queue in the connection descriptor. Whenever the connection thread eventually runs, it begins to process the data that has been delivered between the time of the call of the thread creation and its actual first execution (although note that in Solaris the overhead to create a thread is low).

Each thread has two associated queues: an *event queue* and a *data queue*. The data queue is a list of atm cells. Each connection thread waits blocking on its event queue; it only wakes up if another thread puts an event into its queue. In that case it might take data out of the data queue, process it and make calls to the ATM layer and/or unblock a waiting application thread.

When a connection thread is started, its connection descriptor already exists and so does the entry in the active connection table. The connection thread blocks on its event queue, waiting for a notification that it has something to do. The primary messages it will receive are timeouts on unacknowledged packets, and packet arrival messages. Whenever the connection thread wakes up, it can why it woke up from the type of event in the message queue.

If the connection thread woke up due to a timer event, then it will have to re-send the unacknowledged packet and reschedule the timer event.

When the atm reader thread receives an atm cell for this connection, it puts the cell into the incoming cell queue of the connection through a call to `upcall_deliver_cell()`. This call does not send an event to the connection thread unless the cell is the last cell of a packet (i.e. PT=1).

If the connection thread woke up because a new packet has arrived, it will use the checksum to see if it is valid (no copy is performed in this step). If it is valid, it will check the flow control information to see if any heretofore unacknowledged packets can now be acknowledged. If so, the timeout events for those packets will be descheduled and the packets will be discarded. If the flow control information allows further packets to be sent, then any unsent data packets will be sent (up to windowing limits). These packets will then be placed into the unacknowledged packet queue, and respective timer events scheduled.

After dealing with all pending events, the connection thread will wait again blocking on the event queue.

When a user thread writes data (i.e., makes a `aal7_send()` call), the user buffer will be segmented and copied into the data fields of ATM cells. The AAL7 trailer will be built, and also put into the data fields of ATM cells. By allocating only complete ATM cells, including the ATM headers contiguous to the data fields, we are facilitating later attempts to avoid all unnecessary copying.

Once the calling thread has such a segmented packet, it then checks to see if flow control will allow that packet to be sent immediately. If so, the packet is put on the queue for the Writer Thread and is placed into the queue of unacknowledged packets. If it cannot be sent yet, the packet is placed into the queue of unsent packets. If a data packet was sent, a timeout is scheduled for retransmission if the expected acknowledgement does not arrive.

For early stages of implementation we are using dynamically allocated queues. Eventually, we will dedicate a certain amount of memory to each AAL7 connection. Since all of the data held by the connection is located in queues of lists of cells, we need only count the total number of cells used to determine the amount of memory used. The number of new free cells that this thread can get (say for incoming data) will be the number of cells it is below it's limit.

5.3 Events in the AAL7 layer

All AAL7 connection threads are waiting blocking on their respective event queues. There are data associated with each event; for the exact description of how events are processed see Section 1.5.

- `AAL7_EV_APPLICATION_DISCONNECT`

  This denotes that a function call to `aal7_disconnect()` has been made. There is no data associated with this event. The exact description of what happens can be found in the description of `aal7_disconnect()`.

- `AAL7_EV_ACK_TIMEOUT`

  This event denotes a timeout on an ACK that was not received in time. The data of the event are two 32-bit integers, the first and the last sequence numbers of the associated packets. This allows to schedule one timeout for several packets, as long as the packets have continuous sequence numbers. This avoids expensive kernel calls to wait on a condition variable. Since an AAL7 thread can enqueue through one downcall (`atm_c::sendCellList()`) a (theoretically) arbitrarily large number of atm cells into the Write Queue (and so a large number of packets), a "range-timeout" for these packets seems logical.

- `AAL7_EV_ATM_DISCONNECT`

  This denotes that the ATM layer has either received a disconnect from the other side or that an error occurred in the ATM layer and the connection was torn down (e.g., a broken connection could not be reestablished due to a network partition). The data (two integers) associated with this event contains two integers: first the reason why the ATM layer closed the connection (either due to an error or due to a disconnect from the other side) and also an error code that specified the specific reason of an erroneous disconnect.

- `AAL7_EV_LAST_CELL`

  This event denotes that the last cell of an AAL7 packet has arrived in the data queue of the connection; the thread is woken up to process the packet. There is no data associated with this event.

- `AAL7_EV_TERMINATE`

  This event is only used if there has been a call to `aal_term()` from the application layer. The connection threads free their resources and terminate.

5.4 The AAL7 Service Interface

Note that there are no daemon threads in the AAL7 layer.

- `aal7_init()`

  This function initializes the three shared tables (the active connection table, the listen table and the type table). Then it makes a call to `atm_init()` where it passes on the arguments.

- `aal7_term()`

  Terminates all all7 connection threads by putting the event `AAL7_EV_TERMINATE` into their event queues. Then it makes a call to `atm_term()`.

- `aal7_listen()`

  This call checks first if the SAP that is specified is already in use and if there is enough space in the listen table; these checks can be performed by checking exclusively in the listen table. Then an entry in the listen table is created (and herewith a synchronized queue of virtual channel identifiers, the listen queue). The index of the new entry in the listen table is returned as the local descriptor of the established SAP.

- `aal7_accept`

  This blocks the application thread on the listen queue of the specified SAP. Whenever there is an entry in the listen queue (i.e., an upcall `upcall_connect_request` from the atm layer put a virtual channel identifier into the queue), it spawns a new thread that will handle the connection. At this point the application thread returns.

- `aal7_send()`

  In this function, first the user buffer is segmented into 48 byte pieces that are copied into 53 byte buffers (the ATM cell size). Then the entries for the credit schema in the connection descriptor are checked whether it is possible to directly send new packets. Now one or both of the following two cases can arise: (i) the credit information allows for some packets to be sent immediately, and (ii) the credit information does not allow for some packets to be sent immediately. In case (i), as many packets as can be sent immediately are built, the corresponding AAL7 trailers are appended and the packets are sent through a call to atm_c::sendCellList(). A timeout event for the corresponding connection thread is scheduled. Then the credit information in the connection descriptor is updated accordingly and pointers to the packets are inserted into the unacknowledged packets list. Note that in case (i), the connection thread is not woken up; instead the application thread obtains a lock on the connection descriptor and does all necessary data manipulation. In case (ii), the atm cells are appended to the unsent cells list in the connection descriptor and the application thread returns. In this case, no packets can be sent unless information from the other end of the connection changes the credit information.

- aal7_recv()

  If the received packets list in the connection descriptor of the connection is empty, the thread block on the condition variable in the connection descriptor. Whenever it is unblocked, either the connection broke down or at least one packet has arrived. In case the connection broke down, the application thread returns with status AAL7_STAT_UNREACHABLE. If at least one packet arrived, the application thread copies the specified number of bytes from the packet into the user buffer, then it gives any freed atm cells to the free cell handler object, updates the receive offset in the connection descriptor and returns.

- aal7_disconnect()

  This is a blocking call that waits until all outstanding data that have not yet been sent are sent and acknowledged. There are two cases. In case (i), there are no unacknowledged data, in case (ii) there are. Both cases are handled in a uniform way. The application thread puts an AAL7_EV_APPLICATION_DISCONNECT event into the event queue of the connection thread and then waits blocking on the condition variable in the connection descriptor. All connection termination handling is left to the connection thread including making sure that all unsent packets will be sent and acknowledged. As soon as the connection thread has "flushed" its unsent data and received the corresponding acknowledgements, it removes the entry for its connection from the active connection table, signals the application thread and then terminates. The application layer thread then deletes the connection descriptor. Note that all data are cleaned up and there is no conflict with who deletes what: the connection thread deletes the entry from the active connection table but not the connection descriptor. Then the connection thread terminates without accessing any shared data. So even if the application thread gets rescheduled immediately after receiving the signal from the connection thread, it can safely delete the connection descriptor.

5.5 The AAL7 protocol

5.6 Active Connection Establishment

In an active connection establishment, the application layer makes a call to aal7_connect(). In the following section, we explain each state and the possible state changes in detail. The corresponding state diagram is depicted in Figure 15.

- CLOSED This state is the beginning state of a connection. This state can only be left through a call to aal7_connect() from the application layer (note that we are currently focussing on an active connection establishment).

- ATTEMPTING CONNECTION In this state, function calls to the ATM layer function atm_c::connect() are made. A call returns either ATM_CALL_OK if a connection could

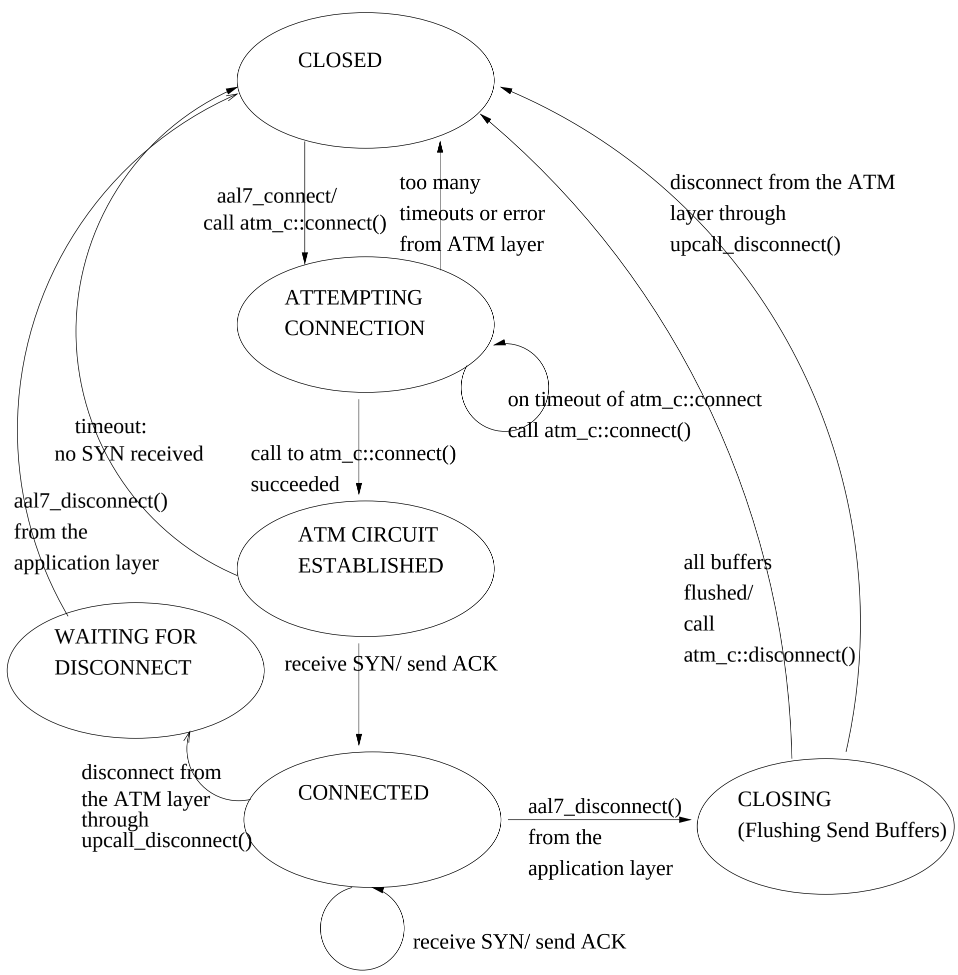

Figure 15: AAL7 Active Connection Establishment

be established; in this case the protocol moves to state ATM CIRCUIT ESTABLISHED. If the call returns ATM_CALL_TIME_OUT, for a constant number of tries a new connection establishment (i.e., a call to `atm_c::connect()`) is attempted. If too many timeouts occur or if the call to the ATM layer returns with another error than a timeout, the connection is closed.

- ATM CIRCUIT ESTABLISHED In this state the ATM layer has a circuit established. We are waiting for a SYN from the other end. If we timeout, we return to the CLOSED state. If we receive a SYN, we send an ACK and go to the CONNECTED state.
- CONNECTED In the CONNECTED state, we might receive a SYN from the other side in case the ACK that we sent at the transition from ATM CIRCUIT ESTABLISHED to CONNECTED is lost. In that case we just retransmit the ACK. Note that with this ACK already a first packet of data can be sent. If we receive an upcall from the ATM layer `atm_upcall_disconnect()`, we go to the state WAITING FOR DISCONNECT. If we receive a downcall from the application layer `aal7_disconnect()`, we might have to send all unsent and unacknowledged packets, so we make a transition to CLOSING.
- WAITING FOR DISCONNECT In this state, we have already received an upcall from the ATM layer that the connection from the other side has closed (either due to a network partition or due to a disconnect from the other side). But the application layer on top might still want to receive any packets that have not been handed over yet to the application space. Therefore we maintain this state in which calls to `aal7_recv()` from the application layer can still be handled properly. Whenever the application layer calls a `aal7_disconnect()`, we go to the CLOSING state.
- CLOSING In the CLOSING state, we are only concerned with flushing our own buffers; we advertise a receive window of zero and try to send all unsent and unacknowledged packets. Otherwise we will be able to flush our buffers eventually; in this case we make a downcall `atm_c::disconnect()` and return to the CLOSED state.

5.7 Passive Connection Establishment

In a passive connection establishment, the application layer makes first a call to `aal7_listen()`. In the following section, we explain each state and the possible state changes in detail. The corresponding state diagram is depicted in Figure 16. Note that we only depict the states for opening and closing a connection, not for when the state is CONNECTED.

- CLOSED This is the beginning state of a connection (note that we are currently focussing on a passive connection establishment). Through a call to `aal7_listen()` we go to the state LISTENING.
- LISTENING In the LISTENING state, two different things can happen. Either an upcall from the ATM layer `upcall_connect_request()` requests a connection. This request is placed into the listen queue of the entry in the listen table of the current connection. In this case we make a transition to the state LISTEN PENDING. If we receive a downcall from the application layer (`aal7_accept()`), the application thread waits on the listen queue of the SAP for a remote connection request to arrive. In this case we make a transition to the state WAITING FOR REQUEST.

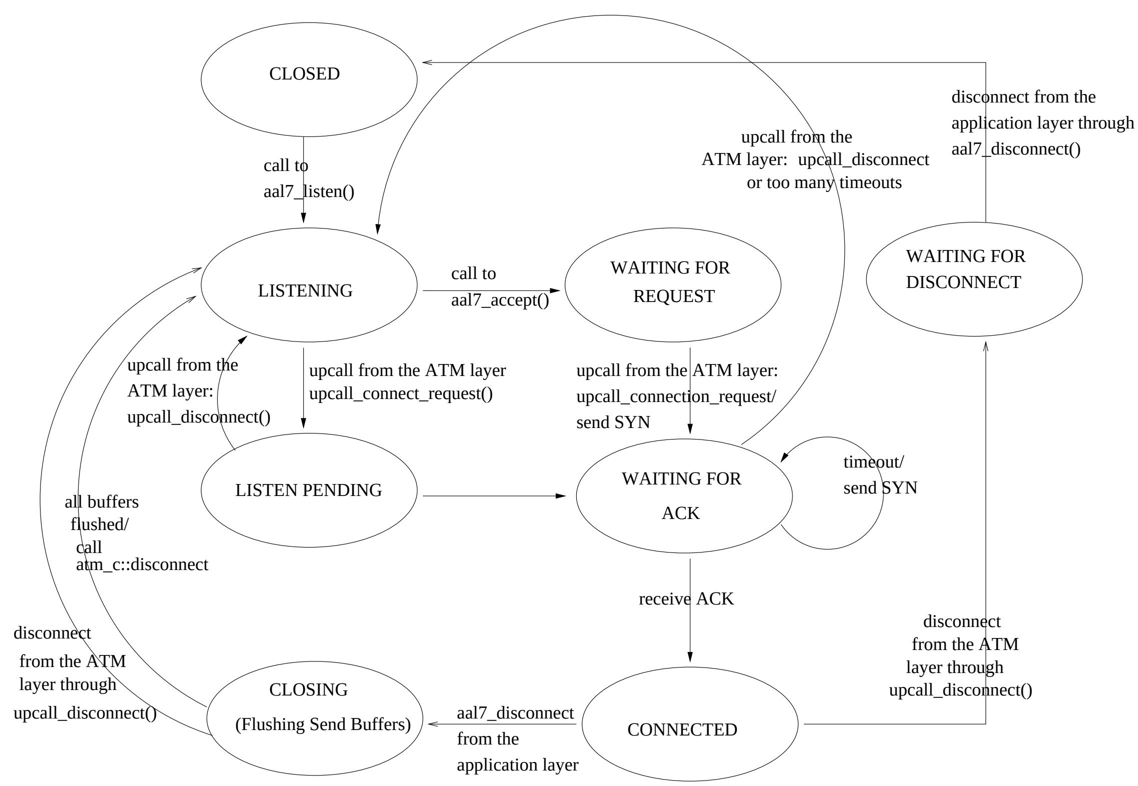

Figure 16: AAL7 Passive Connection Establishment

- LISTEN PENDING In this state an active connection request is waiting in the listen queue to be accepted by the application layer. If the application layer makes a downcall to `aal7_accept()`, we send a SYN and make a state transition to the WAITING FOR ACK state.

- WAITING FOR REQUEST In this state a thread from the application layer is blocked on the listen queue through a call to `aal7_accept()`. Since a call to `aa7_accept()` blocks (theoretically) for an infinite time, we can only transit to another state through an upcall from the ATM layer, the upcall `upcall_connect_request()`. In this case we send a SYN to the other side and transit to the state WAITING FOR ACK.

- WAITING FOR ACK In this state we are waiting for an ACK of the SYN that we sent (note that in both cases in which we entered the state WAITING FOR ACK we just sent a SYN). If we do not receive an ACK, we try for a constant number of times we have to retransmit the SYN; if we do not eventually receive the ACK we return to the CLOSED state. We also return to the CLOSED state if we receive an upcall `upcall_disconnect()` from the ATM layer. Otherwise we receive an ACK and make a transition to the CONNECTED state.

- CONNECTED, WAITING FOR DISCONNECT, CLOSING

  These states are similar to the corresponding states in the active connection establishment. The only difference is once a call to `aal7_listen()` has been made, this SAP is established and can not be "unlistened". Therefore the transitions from CLOSING, WAITING FOR DISCONNECT and CONNECTED go to the state LISTENING instead of the state CLOSED.

5.8 Flow-control mechanism

We will implement a standard sliding window protocol; we acknowledge whole packets. The acknowledgements will be cumulative. All connections will start with the same sequence number (zero) on both the sending and receiving side and they will have a standard receive window size. The acknowledgement number in the AAL7 trailer will be the number of the next packet that the receiver is willing to receive (this is the same schema as in TCP, except that we acknowledge packets, not bytes). All packets, except packets that do not contain data (i.e., have a length field of zero), need to be acknowledged (note that there is one exception to this rule that is explained in the next paragraph). The credits field of an AAL7 trailer will contain the number of packets that the receiver is willing to receive, starting with the ACK that is given in the packet.

In the connection establishment the passive connection end sends one empty packet with a SYN to the active connection end. A SYN is encoded in the first bit of byte 12 in the AAL7 trailer (note that the first three bits of byte 12 are labeled as unused in the AAL7 trailer and we therefore took the liberty to make use of one of them), i.e., normally this bit is set to zero except for a packet that carries a SYN in which the bit is set to one. A packet that contains a SYN is the only case where a packet of length zero needs to be acknowledged.

In the AAL7 trailer, we only make use of the first of the three unused bits; the other two bits are ignored. Whenever the ACK bit is set, the contents of the acknowledgement number and the credits field are interpreted; otherwise both are ignored.

6 Schedule

6.1 Task Breakdown Among Group Members

Module	Implementor	Deadline	Testor
Public Interfaces (headerfiles)	Everybody	done	
hline Basic Init and Terminate	John	done	Johannes
ATM read thread	Krzysztof	done	John
ATM write thread	Krzysztof	done	John
ATM vc thread part 1	John	partially functional	Johannes
Synchronized queue class	Johannes	done	Krzysztof
ATM routing: static	Krzysztof	done	Johannes
Timer class	Johannes	11/19	John
ATM vc thread part 2	John	11/26	Johannes
ATM routing: dynamic	Krzysztof	11/20	Johannes
AAL7: Initialization	Johannes		John
AAL7: Send and receive	Krzysztof	11/22	John
AAL7: Flow control	Johannes	11/	Krzysztof
Garbler and ftp integr.	John	12/04	everybody

6.2 Naming and Programming Conventions

- C++
- Use `assert` whenever feasible.
- Use lower case names with underscore separating words.

- Booleans end in `_b`, classes in `_c`, types in `_t`, enumerated types in `_e`.
- Minimize use of public variables, use accessor method's instead.
- All code is maintained in joint CVS directory. Only commit code that doesn't break compiler.

6.3 Performance Enhancement Strategies

- Data is only copied to and from Application layer space.
- We may replace `mutex_t` with reader writer locks.
- We pass lists of cells from adaptation layer to ATM to DLL, reducing context switch and synchronization overhead.
- Free `atm_cell_list` avoids new/delete.
- We partition the VCI's according to first 2 bits to allow for rapid discrimination between cells in hardware.
- We will experiment with timeout values.
- We may fine tune the thread/LWP bindings.
- We intend to implement hash tables and other quick look up schemes for mappings.
- We will experiment with free cell list management.