Extending ATM Networks for Efficient Reliable Multicast

Jonathan S. Turner
jst@cs.wustl.edu

WUCS-96-16

January 13, 1997

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

One of the important features of ATM networks is their ability to support multicast communication. This facilitates the efficient distribution of multimedia information streams (such as audio and video) to large groups of receivers (potentially millions). Because ATM networks do not provide reliable delivery mechanisms, it is up to end systems to provide end-to-end reliability where it is needed. While this is straightforward for point-to-point virtual circuits, it is more difficult for one-to-many and many-to-many virtual circuits. In this report, we propose some minimal extensions to the hardware of ATM switches that enables end systems to implement reliable multicast in a more efficient and scalable manner than is otherwise possible. Our approach makes the amount of work that must be done by any end system essentially independent of of the number of multicast participants, while requiring no complex processing at the switches. We propose solutions for both one-to-many and many-to-many multicast virtual circuits.

[0]This work was supported by the Advanced Research Projects Agency.

Extending ATM Networks for Efficient Reliable Multicast

Jonathan S. Turner
jst@cs.wustl.edu

1. Introduction

Efficient multicast communication is a key feature of ATM network technology. Originally designed to support distribution of multimedia information streams (audio and video), it can also be useful for more general distributed computing applications. However, because such applications generally require reliable data delivery, end systems must provide mechanisms for ensuring reliable transport over the unreliable communication channels provided by ATM networks. While providing reliable transport is straightforward for point-to-point communication channels, it is more complex in the context of multicast channels which may have hundreds, thousands or even millions of participants.

This paper develops an approach to reliable multicast communication in ATM networks. Our objective is to provide *scalable* mechanisms by which we mean mechanisms in which the amount of work done by senders and receivers in a multicast connection is independent of the total number of participants. Reliable multicast can be implemented entirely by mechanisms in end systems or by some combination of mechanisms in end systems and the network. In addition to being senders and receivers, end systems can also act act as multicast servers. In this paper however, we use end systems primarily as senders and receivers although we also consider solutions to the many-to-many multicast problem that use an end-sytem as a relay in a many-to-many connection. Our restrictions on the role of end systems are motivated by our desire to provide the highest possible performance and the lowest possible latency.

There is a substantial literature on providing reliable multicast mechanisms in networks. One of the earliest contributions was [4]. More recent contributions include [1, 2, 6, 8, 10, 9, 11, 12, 13]. Much of this more work is directed toward multicast in the context of the internet protocol suite and shifts much of the responsibility for recovering lost data onto the receivers. The recent paper by Floyd, Jacobson, et. al. [7] is a good example of the receiver-based recovery. This paper also advocates the use of application-specific reliable multicast mechanisms, rather than more general mechanisms supported at lower levels. While some of the prior work is applicable in the ATM context, there seems little prior work directly related to extending ATM switching mechanisms to improve support for reliable multicast.

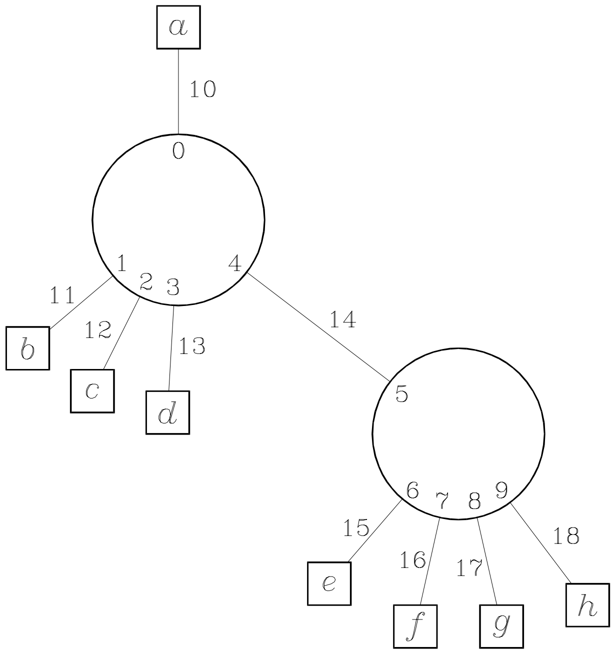


Figure 1: Reliable Multicast Message Transmission

The paper has three main parts. The first addresses one-to-many multicast and shows how a simple extension to ATM switching systems can be used to provide reliable multicast mechanisms that exhibit excellent scalability. The second part addresses the problem of many-to-many multicast, considering several different approaches. Finally, we give detailed design information for the implementation of these mechanisms in the Washington University Gigbait Switch.

2. One-to-Many Multicast

As indicated in the introduction, we are interested in reliable multicast mechanisms in which the amount of work that must be done to send and/or receive a packet by each end system participating in the multicast is independent of the total number of participants. In the case of one-to-many connections, this means that the amount of work the sender must do to send a packet reliably (including processing the acknowledgements) should be the same, whether there is a single receiver or a million. We also want the work done by receivers to be independent of the number of participants. Thus, when it is necessary to retransmit a packet because not all participants received it, we want the retransmission to go to only those receivers that failed to receive the original transmission.

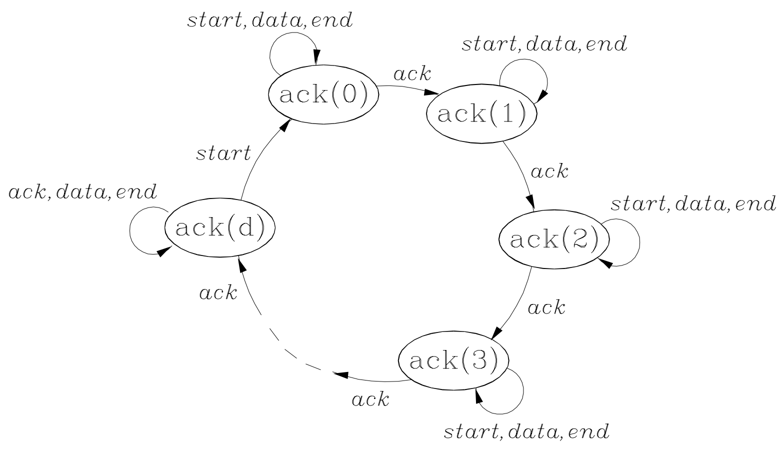
2.1. Hardware Assist Mechanisms for One-to-Many Multicast

Our approach to achieving reliable one-to-many multicast is illustrated in Figure 1. To send a packet to end systems b through h, end system a sends the packet on a reliable multicast virtual circuit, through the two ATM switching systems, shown in the figure. The packet is delineated by start and end of packet cells (these, and the other control cells required, can be implemented using a variant of the ATM Resource Management cell type). The ATM switches note the passage of the start and end cells, and when acknowledgement cells are received from the downstream neighbors, they propagate only the last acknowledgement cell expected. The acknowledgement cells are originated by the receiving end systems; the switches merely propagate them selectively, so that the sender receives only one acknowledgement indicating that all destinations have received the message. If the acknowledgement is not received until after a timeout has expired, source a can send the packet again, preceded by a new start cell. The switches propagate the retransmitted packet, only to those downstream neighbors that did not acknowledge the original transmission, so that destinations that received the first copy will not have to process a duplicate.

To allow end systems to pipeline multicast packet transmissions through the network, all control cells processed by the switches (start, end, ack) contain *transmission slot numbers* which are used to access stored state information relevant to a particular packet. Individual data cells do not contain slot numbers, but the data cells are assumed to be sent using AAL 5 (or something equivalent) and include the slot number or an equivalent transport protocol sequence number somewhere in the end-to-end protocol header. A receiver acknowledges a packet only if it is correctly received (as indicated by an end-to-end error check).

The switches maintain a state machine for each transmission slot, to keep track of which downstream neighbors have acknowledged a given packet and which have not. One version of this state machine is shown in Figure 2. In the state diagram, the state $ack(i)$ denotes all states in which i acknowledgements (out of a total expected number of d) have been received. Assuming the state machine starts in state $ack(d)$, the arrival of a start cell, places it in state $ack(0)$. In this state it forwards data cells and the end cell. When the first acknowledgement is received, it notes which downstream neighbor acknowledged the cell and proceeds to state $ack(1)$. Additional acknowledgements trigger further transitions. The transition to state $ack(d)$ also triggers the forwarding of an acknowledgement to the upstream neighbor. When in state i for $i < d$, new start, data or end cells are simply forwarded to those downstream neighbors that have not yet acknowledged the packet. The program fragment in the figure shows the processing that would be done at a typical switch in a reliable multicast connection in response to the various types of cells. Note that `status[j]` need not really be a separate variable, since its value is implied by the value of `ackset[j]`. We've chosen to show it separately only for clarity of exposition.

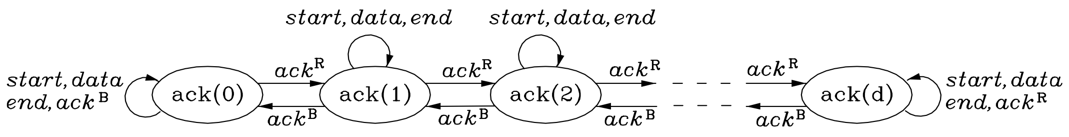
Unfortunately, the state machine in Figure 2 is deficient in two respects. First, consider what happens if a given switch completes a packet and goes to state $ack(d)$, sends its acknowledgement to its upstream neighbor, and then the acknowledgement is lost before it reaches the upstream neighbor. Eventually, the sender will retransmit the packet beginning with a new start cell. The switch whose acknowledgement was lost should recognize this situation and simply convert the start cell to an acknowledgement cell and return it, discarding the subsequent data and end cells. However, the state machine as written, will treat this as a new transmission and forward it on to the downstream neighbors. Sequence numbers in the transport level packet can prevent the receiving end systems from being confused by this redundant transmission, but they will be forced to do some unnecessary



```
upstream_neighbor = input link and VCI on which packets arrive
downstream_neighbor = link and VCI on which a received ack arrived
output_set = set of output (link, VCI) pairs to which packets
             are to be forwarded
type = type of the cell being processed (data, start, end, ack)
i = slot number in the control cell being processed
    (i is undefined if processing data cell)
currslot = slot number in last start cell processed
status[j] = status of slot j (options are ack(0) ... ack(d))
ackset[j] = subset of downstream_neighbors that have ack'ed slot j

if type = start and status[i] = ack(d) then
    status[i] = ack(0)
    ackset[i] = {}
    currslot = i
    forward cell to output_set
if type = start and status[i] != ack(d) then
    currslot = i
    forward cell to (output_set - ackset[i])
if type = data then
    forward cell to (output_set - ackset[currslot])
if type = end then
    forward cell to (output_set - ackset[i])
if type = ack then
    ackset[i] = ackset[i] + {downstream_neighbor}
    update status[i]
    if status[i] = ack(d) then
        forward ack to upstream_neighbor
    else
        discard cell
```

Figure 2: State Machine for Monochromatic Algorithm

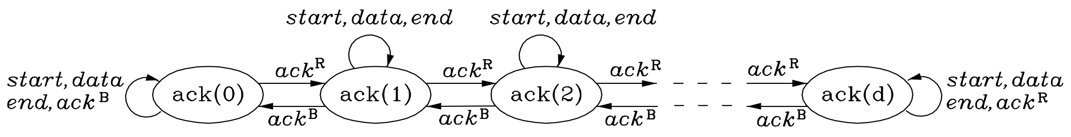


```
if status[i] = ack(0) then
    if type = start and cell_color = red then
        ackset = {}
        currslot = i
        currcolor = red
        forward cell to output_set
    else if type = start and cell_color = black then
        convert cell to ack and return to upstream_neighbor
    else if type = data or type = end then
    else if type = ack and cell_color = red then
        ackset[i] = ackset[i] + {downstream_neighbor}
        discard cell
    else if type = ack and cell_color = black then
        discard cell
else if status[i] = ack(d) then
    if type = start and cell_color = black then
        ackset = output_set
        currslot = i
        currcolor = black
        forward cell to output_set
    else if type = start and cell_color = red then
        convert cell to ack and return to upstream_neighbor
    else if type = data or type = end then
        if currcolor = black
            forward cell to output_set
        else
            discard cell
    else if type = ack and cell_color = black then
        ackset[i] = ackset[i] - {downstream_neighbor}
        discard cell
    else if type = ack and cell_color = red then
        discard cell
```

Figure 3: State Machine for Bichromatic Algorithm

work. The more important problem with this approach however, is that its correctness requires that the sender never initiate a retransmission if there is any possibility of the receiver still sending an acknowledgement. There are cases where the combination of a slow receiver, lost control cells and unlucky timing can result in the sender thinking a packet has been correctly received, when in fact, it has been lost.

To correct this problem, we add a *color bit* to the slot numbers carried in the various control cells. We require that senders alternate the color of consecutive packets sent with the same slot number. This leads to the state machine shown in Figures 3 and 4. Here,



```
else if status[i] = ack(1) or . . . or status[i] = ack(d-1) then
    if type = start then
        currslot = i
        currcolor = cell_color
        if currcolor = red then
            forward cell to output_set - ackset[i]
        else
            forward cell to ackset[i]
    else if type = data or type = end then
        if currcolor = red then
            forward cell to output_set - ackset[i]
        else
            forward cell to ackset[i]
    else if type = ack and cell_color = red and currcolor = red then
        ackset[i] = ackset[i] + {downstream_neighbor}
        if ackset[i] = output_set then
            forward ack to upstream_neighbor
        else
            discard cell
    else if type = ack and cell_color = black and currcolor = black then
        ackset[i] = ackset[i] - {downstream_neighbor}
        if ackset[i] = {} then
            forward ack to upstream_neighbor
        else
            discard cell
    else
        discard cell
```

Figure 4: Bichromatic Algorithm (cont.)

`cell_color` represents the color bit of the control cell being processed and `currcolor` represents the color bit of the most recent start cell. In this algorithm, the state machine is in either the leftmost or rightmost state when propagating a packet for the first time. As acknowledgements of the proper color are received, the state machine moves to the other end. Thus, when processing red packets, `ack(i)` designates states in which `i` acknowledgements have been received, while when processing black packets, it designates states in which `i` acknowledgements are still expected. To distinguish this algorithm from the original, we refer to it as the *bichromatic* algorithm and the original as the *monochromatic* algorithm. Note that the bichromatic algorithm correctly handles lost acknowledgements and its correctness is independent of timing considerations [1]

[1]The author thanks Andy Fingerhut for clearly identifying the deficiencies in the monochromatic algorithm and showing how they could be corrected without adding any state.

2.2. A One-to-Many Multicast Transport Protocol

In this section, we introduce a conventional sliding window protocol to provide reliable multicast transmission, in conjunction with the switch level mechanisms described in section 2.1. The transport protocol is a conventional sliding window protocol. An implementation of the protocol (expressed in the Java programming language) is given in Figures 5 and 6. The implementation comprises two classes, a `sender` class and a `receiver` class. Each class has a constructor which initializes the essential data, and one or more `synchronized` methods, which are executed atomically and exclusively by the different threads that invoke the methods.

Figure 5 shows the sender class. The `SendData` method is invoked by a the user of the transport protocol with data to send. If necessary, the user is suspended until a buffer slot is available to store the user's data. When a buffer is available, the user's data is placed in a packet with an appropriate sequence number and passed to the driver. The driver sends the packet, on the ATM virtual circuit (we've omitted the passing of the ATM level information to the driver), preceded by a start cell and followed by an end cell each containing a slot number equal to the sequence number in the packet, modulo the window size, `W` and a color bit which is red if the integer quotient of the sequence number with `W` is even, and black otherwise. It then sets a timer which will trigger a retransmission if no acknowledgement is received in the time expected. The protocol is designed to require only enough buffers for the packets in the window. This works because the sender only allows `W` packets with sequential sequence numbers to be in transit at one time. Similarly, the `Acked` array need only be dimensioned for the packets in the window and the slot numbers sent in the start and end cells can be limited to `W` distinct values without ambiguity. (Note however, that it is necessary that `N` be an even multiple of `2*W`.)

When an acknowledgement is received, the physical layer driver invokes the `HandleAck` method of the sender with the sequence number from the acknowledgement cell. If the sequence number of the received ack is within the current window limits, the sender sets the appropriate bit in the `Acked` array and stops the timer. It then advances the low end of the window past all buffer slots that have now been acknowledged, making those buffer slots available to new user data blocks. `HandleAck` ends by checking if there are any available buffer slots and if so invokes the `notify` method, waking up a potentially waiting user thread.

The `HandleTimeout` method is invoked by the `timerset` object when a timer expires. The sequence number associated with that timer is passed to the method. This triggers a retransmission of the packet. When the method returns to the `timerset` object it restarts the timer automatically.

Figure 6 shows the receiver class for the transport protocol. Its `HandlePacket` method is invoked by the driver when the driver receives a sequence of data cells forming a correct packet. The driver also sends an ack for every packet correctly received. Each ack contains both the sequence number of the received packet and the transmission slot number. The `HandlePacket` method checks to see if the received packet is within the current receive window and has not yet been received. If so, it stores the packet, sets the `Arrived` bit and then passes as many received packets as it can to the user.

```
class sender {
    private int       W;          // window size
    private int       N;          // number of sequence numbers N%(2*W)==0
    private int       slo;        // low end of send window
    private int       shi;        // high end of send window
    private int       TimedOut;   // sequence number of frame that timed out
    private usrdata[] OutBuf;     // array of outgoing data blocks
    private boolean[] Acked;      // Acked[i] = true if OutBuf[i] has been
                                  // acknowledged
    private int       w;          // number of blocks in send window currently
    private dprot     driver;     // object implementing physical link driver
    private timerset  timers;     // object implementing set of timers

    void sender(int W1, int N1, dprot driver1, timerset timers1) {
        W = W1; N = N1; driver = driver1; timers = timers1;
        slo = shi = w = 0;
        OutBuf = new usrdata[W];
        Acked = new boolean[W];
        for (int i = 0; i < W; i++) Acked[i] = false;
    }
    synchronized void SendData(usrdata ud) {
        packet p;
        if (w == W) wait();     // wait for an empty buffer slot
        p.info = OutBuf[shi % W] = ud;
        p.seq = shi; Acked[shi % W] = false;
        w++; shi = (shi + 1) % N;
        driver.SendPacket(p);   // driver adds start and end cells
        timers.StartTimer(p.seq);
    }
    synchronized void HandleAck(int ack) {
        if ((shi >= slo && (ack >= slo && ack < shi)) ||
            (shi <  slo && (ack >= slo || ack < shi))) {
            Acked[ack % W] = true;
            timers.StopTimer(ack);
            while (w > 0 && Acked[slo % W] ) {
                w--; slo = (slo+1) % N;
            }
            if (w < W) notify();    // wakeup waiting user thread, if any
        }
    }
    synchronized void HandleTimeout(int TimedOut) {
        packet p;
        p.info = OutBuf[TimedOut % W];
        p.seq = TimedOut;
        driver.SendPacket(p);
    }
}
```

Figure 5: Sender for Transport Protocol

```
class receiver {
    private int       W;          // window size
    private int       N;          // # of sequence numbers N%(2*W)==0
    private int       rlo;        // low end of receive window
    private int       rhi;        // high end of receive window
    private usrdata[] InBuf;      // array of incoming packets
    private boolean[] Arrived;    // Arrived[i] = true if expected packet
                                  // for InBuf[i] has been received
    private uprot     user;       // object using the sender

    void sender(int W1, int N1, uprot user1, dprot driver1) {
        W = W1; N = N1; user = user1; driver = driver1;
        rlo = 0; rhi  = W;
        InBuf = new usrdata[W];
        Arrived = new boolean[W];
        for (int i = 0; i < W; i++) Arrived[i] = false;
    }
    synchronized void HandlePacket(packet p) {
        if (((rlo <= rhi && (rlo <= p.seq && p.seq < rhi)) ||
            (rlo >  rhi && (rlo <= p.seq || p.seq < rhi))) &&
            !Arrived[p.seq % W]) {
            Arrived[p.seq % W] = true;
            InBuf[p.seq % W] = p.info;
            while (Arrived[rlo & W]) {
                user.DeliverData(p.info);
                Arrived[rlo % W] = false;
                rlo = (rlo + 1) % N;
                rhi = (rhi + 1) % N;
            }
        }
    }
}
```

Figure 6: Receiver for Transport Protocol

The combination of the bichromatic algorithm described in section 2.1 with the sliding window transport protocol is sufficient to ensure reliable multicast transport, assuming that any transmission errors in either control cells or data are detected by error-detecting codes triggering discarding of the faulty data. To see this, note that since sequence numbers are assigned modulo N with $N \geq 2W$, N is a multiple of $2W$ and the windows at sender and receivers contains packets with consecutive sequence numbers, there is no possibility for sender and receivers to identify packets inconsistently, nor is it possible for switches to identify control cells inconsistently. This ensures that no packet is delivered to the user at any receiver more than once. Note that the loss of an end cell and the subsequent start cell just upstream of a given switch can cause that switch to effectively combine two packets together into one; however the end-to-end error check ensures that the receivers will detect the error and reject the received data in such situations, forcing retransmission of both affected packets. Since the sender continues to retransmit a packet until all receivers acknowledge it, and the acknowledgement is received by the sender, every packet is eventually delivered so long as no link corrupts any specific packet infinitely many times.

Note that these mechanisms do not protect a large multicast connection from the failure of one of its receivers. Such a failure can keep packets from being acknowledged, preventing further transmission over a given multicast connection. We don't attempt to address such issues directly within the multicast protocol. However, we note that a sender experiencing repeated timeouts on a reliable multicast connection can resolve the situation by invoking network-level control mechanisms. The ATM network control software can traverse the multicast connection tree examining the status information at various switches to quickly identify the end system (or systems) that have failed to acknowledge a packet on a given connection and either "prune" them from the connection tree or simply inform the sender and allow the sender to decide on the appropriate course of action. Alternatively, one could include a special control cell that would be propagated to all endpoints that have failed to acknowledge a cell with a given slot number and prune them from the connection.

2.3. Incorporating Negative Acknowledgements

The use of timeouts to detect lost packets can delay retransmission unnecessarily, limiting overall performance. Frequently, it's possible for a receiver to detect that a packet has been lost and immediately send a negative acknowledgement to the sender, requesting a retransmission. (In a typical sliding window protocol operating over an "order-preserving" network, the arrival of a packet with a sequence number different from the "next" one in sequence indicates one or more lost packets.) When a packet is lost in a reliable multicast connection, all receivers downstream of the point where the loss occurs may detect the loss and send a negative acknowledgement. To ensure scalability of reliable multicast protocols in the presence of negative acknowledgements, the network should return only the first negative acknowledgement for a particular lost packet. This requires that the switches keep track of whether a nack has been sent for a given packet, and if so to suppress forwarding of further nacks. This requires adding some additional state to the switch state machine.

In particular, for each transmission slot, the switch must keep an additional bit that is cleared when the last positive acknowledgement has been received for a given packet and is set when the first nack for that packet is received. Start cells associated with retransmissions do not re-enable nack forwarding. Referring to the picture at the top of Figure 3, the nack bit would be cleared on a transition from `ack(d-1)` to `ack(d)` or on a transition from `ack(1)` to `ack(0)`. If a negative acknowledgement is received when the nack bit is 0, the nack is forwarded upstream and the nack bit is set, but the state machine of Figure 4 remains in the same state. Note that the correctness of the protocol still relies on the positive acknowledgements; nacks merely improve performance when packets are lost.

2.4. Implementation in Washington University Gigabit Switch

Figure 7 shows the high level organization of the Washington University Gigabit Switch (WUGS). The WUGS switch is constructed from three main components. The *Input Port*

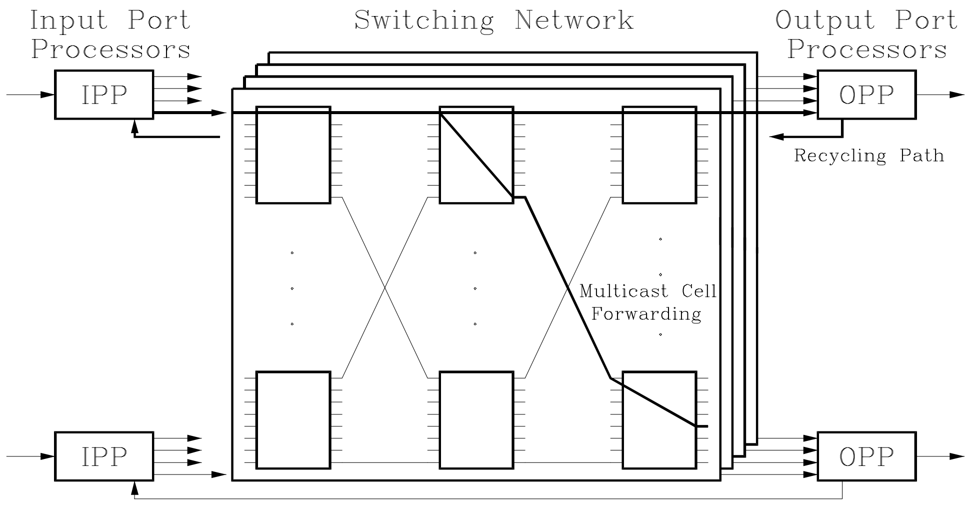


Figure 7: Recycling Switch Architecture

Processors (IPP) at left, receive cells from the incoming links, buffer them while awaiting transmission through the central switching network and perform the virtual path/circuit translation required to route cells to their proper output (or outputs). The *Output Port Processors* resequence cells received from the switching network and queue them while they await transmission on the outgoing link. Each OPP is connected to its corresponding IPP, providing the ability to *recycle* cells belonging to multicast connections. The central switching network is made up of *Switching Elements* (SE) with eight inputs and outputs and a common buffer to resolve local contention. The SEs switch cells to the proper output (or outputs) using information contained in the cell header or can distribute cells dynamically to provide load balancing. The load balancing option is used in the first $k-1$ stages of a $2k-1$ stage network. In particular, for the configuration shown in Figure 7 ($k=2$), load distribution is performed in the first stage. Adjacent switch elements employ a simple hardware flow control mechanism to regulate the flow of cells between successive stages, eliminating the possibility of cell loss within the switching network. With this approach, relatively small buffers suffice within the network. Larger buffers are provided at the OPPs.

The WUGS switch implements multicasting using a technique called *cell recycling*. When cells arrive on an input link, the virtual path and virtual circuit identifiers are used to select an entry from a routing table in the IPP (called the *Virtual Path/Circuit Translation Table* (VXT)). The entry includes a pair of output port numbers, a pair of new virtual path and circuit identifiers and a pair of control bits. The switching network routes a copy of the cell to each of the designated outputs, where they can be forwarded to the outgoing link or optionally recycled back to the input ports (this choice is determined by the control bits), where the virtual path and circuit identifiers are used to perform new VXT lookups, yielding new pairs of destinations. Through this process, a connection with f destinations can be handled in $\log_2 f$ passes through the network. Further details on the WUGS switch architecture can be found in references [5, 14, 15, 16].

Because the WUGS architecture breaks a large multicast into binary copy steps, it can be easily extended to implement the reliable multicast protocol. Each VXT that a cell in a reliable multicast virtual circuit passes through must contain the following information.

- Switch output port numbers for the two downstream neighbors that the cell is to go to next.
- VPI/VCIs for the two downstream neighbors that the cell is to go to next.
- Switch output port number of the upstream neighbor to which acks are to be forwarded.
- VPI/VCI for upstream neighbor to which acks are to be forwarded.
- Maximum slot number that the given connection supports.
- Slot number use by the last start cell processed.
- For each of the `W` packets that can be in transit through the network at one time, a pair of bits specifying the `ackset` plus a `nack` bit.

The first two items in the list are required in any case, but the last five items must be added. Rather than dimension every VXT entry to support reliable multicast, one can implement this feature by allocating a sequence of consecutive VXT entries to a reliable multicast connection as part of the call setup process. The WUGS switch currently provides 18 bytes per VXT entry, so with a single extra entry, reliable multicast connections supporting a window size of 24 can be supported, assuming two bytes for the upstream neighbors' output port number, three bytes for the upstream neighbors' VPI/VCI, two bytes for the maximum slot number and two bytes for the current slot number. Each additional entry allocated to the reliable multicast adds 48 additional slots to the window. For local area connections in which the round-trip delays are small (less than one millisecond), a window size of 24 is sufficient to maintain connection data rates of about 1.5 Gb/s with packet sizes of 8 Kbytes. For wide area connections with a round-trip delay of 40 ms, a window size of 92 is sufficient for connection data rates of up to 150 Mb/s. Thus, most reliable multicast connections can be handled using just a few VXT entries, making this mechanism inexpensive enough to be broadly applicable.

The operation of the reliable multicast mechanism in the WUGS switch is illustrated in Figure 8. The upper part shows how the start cell, data cells and end cell flow through the second of the two switches in Figure 1. Since the connection has a fanout of four at this switch, it makes two passes through the switching network, with the virtual circuit tables at the recycling ports (20 and 21) providing the next pair of port numbers (the VPIs and VCIs are not shown explicitly, but would be included in the actual tables). The lower part of the figure shows how the acknowledgements flow back up the tree, with one extra pass needed at the end, since the VXTs are on the input side of the switch.

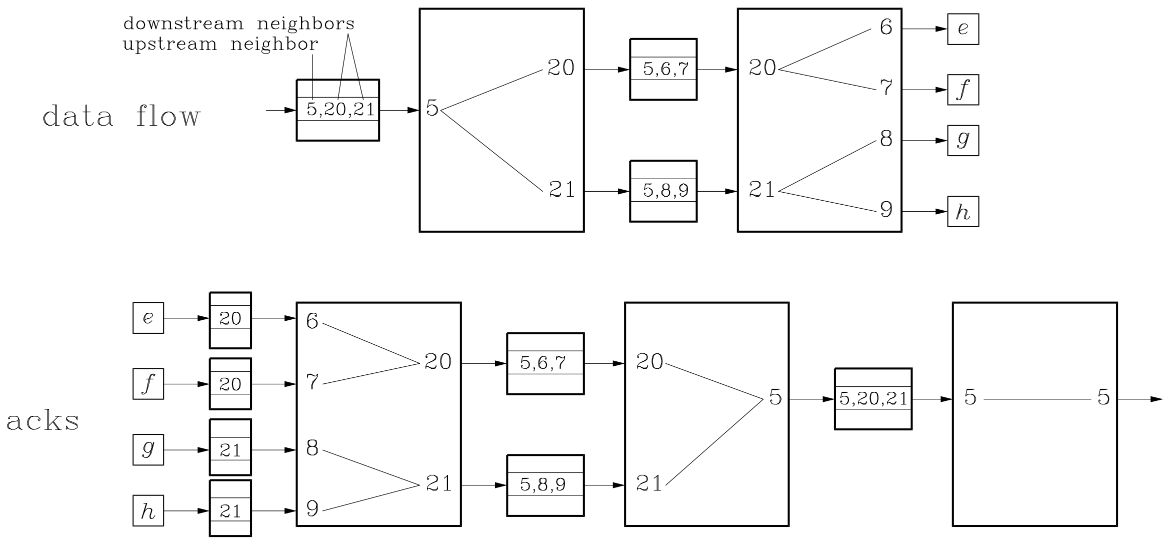


Figure 8: Reliable Multicast Message Transmission

2.5. Performance Analysis

If a packet is delivered to all receivers correctly the first time, the amount of work done by the sender is about the same as for sending a point-to-point packet. Similarly, for each of the receivers. Furthermore, no matter how many times a given packet is lost, the receivers who got it the first time need do no extra work on behalf of other receivers. (In fact, the amount of work done by each receiver is slightly less than with a point-to-point protocol, since a receiver will not receive retransmissions caused by lost acknowledgements, in most cases, whereas with point-to-point protocols, receivers do receive redundant retransmissions in this case.) For the sender, the amount of work per packet transmission is also essentially the same as in a point-to-point protocol. There is no extra work per packet transmission, as a result of having many receivers. However, in large multicast connections, the sender may have to send packets multiple times before they are received at all receivers. The number of transmissions required is really independent of the reliable multicast protocol, but in our case this is the only factor that limits scalability. The remainder of this section assesses the magnitude of this factor.

Consider a multicast connection with n receivers, a maximum sender-to-receiver path length of d links and a probability of packet loss of at most ϵ on any single link. Let x_i be the number of receivers that have not received a given packet following the i-th transmission (we assume that packet losses on different links and in different 'rounds' are independent). For any particular receiver, the probability that it fails to receive a packet when it is first transmitted is $\leq \epsilon d$, hence the expected value of x_1 is $\leq \epsilon dn$. Similarly, the expected value of x_2 is $\leq \epsilon d x_1 \leq (\epsilon d)^2 n$. Similarly, the expected value of $x_i \leq (\epsilon d)^i n$. Since x_i is a non-negative random variable, the probability that x_i is ≥ 1 is less than or equal to its expected value. Using this, one can show that for

$$i \geq \frac{\ln n + \ln 1/\epsilon}{\ln 1/\epsilon d}$$

the probability that $x_i \geq 1$ is at most ϵ. For $n = 1000$, $\epsilon = 10^{-5}$ and $d = 6$, this implies that with probability $1 - \epsilon$, all endpoints receive the packet after two transmissions. For $n = 10^6$, $\epsilon = 10^{-4}$ and $d = 10$, four transmissions are enough. For still larger connection sizes, it might be necessary to have some end systems act as repeaters, buffering packets for retransmission to smaller subsets. However, it's hard to imagine a real application that would require this.

3. Many-to-Many Multicast Through a Relay

In distributed computing applications, it is often desirable to have not just reliable one-to-many communication, but also reliable many-to-many communication, in which any member of a group can reliably send a packet to the other members. Moreover, in some cases it can be important for the packets to be ordered so as to provide consistent delivery order to all receivers. As with one-to-many multicast, we are concerned with ensuring that the amount of work that must be done per packet by each end system is independent of the total number of participants. An important observation, with respect to many-to-many multicasts, is that the total data rate of a multicast is limited by the receivers' capacity to absorb data. Thus, as the number of senders increases, the average data sent by each one must decrease. At the same time, to allow maximum flexibility, we want to avoid constraining how the overall connection capacity is used by the different senders. At one extreme, we can have connections in which all senders transmit continuously at some low data rate. At the other, we can have senders take turns sending at the full connection data rate. Any many-to-many multicast scheme should support both options efficiently, as well as the infinite variety of intermediate possibilities.

There are two key issues that must be addressed in any solution to the many-to-many multicast problem. First, one must deal with interleaving of cells from different sources, so packets can be correctly reassembled at the receivers. Possible solutions within the context of standard ATM include the use of separate virtual circuits, the use of multicast virtual paths with VCIs used for sender identification and the use of AAL 3/4, which includes a Message Identifier (MID) field. Here, we consider solutions that extend standard ATM mechanisms. The second issue that must be addressed is the delivery of acknowledgements from the receivers back to the sender of a given packet. This must be done in a way that is consistent with our scalability objectives.

In this section, we focus on many-to-many multicast mechanisms that use an end system (or network-resident server) as a relay node. This has the advantage of providing consistent delivery order to all receivers. Note that because the total data rate of a multicast connection is limited by the receivers, the use of a relay doesn't limit the throughput of a multicast connection, although it does increase the latency by as much as a factor of two. The use of a multicast relay reduces the problem to two subproblems, a many-to-one followed by a one-to-many. Since the previous section has addressed the one-to-many problem, we focus here on the many-to-one aspect.

3.1. Many-to-Many Multicast Using Point-to-Point VCs to the Relay

The simplest form of many-to-many multicast involves connecting each sender in the connection, via a reliable point-to-point virtual circuit to the relay node. The senders and relay node use a conventional sliding window protocol to transfer data reliably and as soon as a packet is ready to be forwarded at the relay node, the relay passes it across a reliable one-to-many connection using the method described in section 2. The acknowledgement of packets back to the senders can be done either when the packet is correctly received at the relay or after it has been delivered to all the receivers. Alternatively, both types of acknowledgements can be provided.

This approach to many-to-many multicast is quite general. The server can handle any pattern of transmissions from the senders with equal ease, while pipelining data for maximum throughput. The ordering of packets at the relay provides a consistent ordering for all of the receivers. From a processing standpoint, it is fully scalable and can handle connections with large numbers of both senders and receivers. Since there is only a single relay, a higher capacity processor can be used at that point, if need be, to support high traffic volumes. The relay node can be implemented either by an arbitrary end system or by a network-resident multicast server. If reliable multicast is provided as a network service, a server near the geographical center of the set of users can be employed, potentially reducing the latency penalty associated with the use of a relay node.

This approach does introduce a scaling limitation with respect to the amount of protocol state information and buffering that must be maintained at the relay node. Since each sender has its own point-to-point connection to the relay, the memory requirements at the relay must grow in direct proportion to the number of senders. Also, the number of virtual circuits entering the sender must grow in proportion to the number of senders. This limitation poses little problem for the vast majority of applications, but could become an issue for applications with a very large number of senders (more than a few hundred).

3.2. Using Multiple Many-to-One VCs with Centralized Allocation

Most multicast applications involving multiple senders do not require that all senders transmit at the same time. Indeed, the more typical pattern is for one or a few senders at a time to be active, even when the total number of 'potential senders' is very large. This characteristic can be used to produce more scalable many-to-many connections.

Instead of having one virtual circuit for every potential sender, we can have a collection of k many-to-one virtual circuits from the senders to the relay. Each sender can use any one of these virtual circuits to send packets to the relay, but only one sender may use a particular virtual circuit at a time. These k virtual circuits can be managed by having the senders transmit *channel acquisition requests* to the relay, and having the relay assign senders to channels, then inform the senders of its decisions. If the channel acquisition requests are formatted as single cell messages, they can be sent on a dedicated many-to-one control channel without any cell-interleaving problem. Because the relay assigns data channels to single sources, no contention occurs on the data channels, eliminating the cell interleaving problem there, as well. For bandwidth management purposes, the set of many-to-one virtual circuits can be treated as an aggregate, with the bandwidth specification applying to the collection, rather than the individual virtual circuits.

Both the control channels and the data channels require some mechanism by which the relay can send information (channel assignment cells and acks) back to the sources. If we are to avoid scaling limits at the relay, this cannot be done with point-to-point VCs. Similarly, if we are to avoid scaling limits at the senders, we can't just broadcast this information on a one-to-many VC from the relay back to the senders, letting the senders filter out cells addressed to others. What's required is a one-to-many connection from the relay back to the senders, which instead of delivering a cell to all senders, delivers it to a specific sender. This requires some modification of the ATM switch hardware. There are several approaches one can take to this problem, but the simplest appears to be the use of a *trace-back* mechanism, that allows a cell sent in reply to a cell previously received on the many-to-one path to be returned to the same place.

A trace-back mechanism provides a straightforward way to implement the "upstream" part of a many-to-one virtual circuit. It requires a one-to-many virtual circuit that has the same branching structure as the many-to-one virtual circuit that it is used with. Control cells sent on the many-to-one path (i.e. channel acquisition requests, start of packet cells, etc.) include a *trace field* that switches on the path can insert information into. The relay includes the same trace information in cells it sends on the upstream path back to the sender and each switch along the path uses this information to forward the cell to the proper upstream branch. In the case of the WUGS switch architecture described earlier, this process is particularly simple. The many-to-one path is structured as a binary tree and at every point where two branches come together, the switch need only insert one bit into the trace field. This is done most simply by treating the trace field as a stack and "pushing" the new bit onto the "top" of the stack. The one-to-many return path is also structured as a binary tree with the same branching structure as the many-to-one tree. Whenever a cell comes to a branch point in the return path, it selects one of the two branches based on the next bit of the trace field and "pops" that bit off the stack.

3.3. Contention-Based Channel Acquisition

Centralized channel acquisition works well when the time duration of a data transmission from a single sender is longer (on average) than the time required to acquire the channel. In a local network, channel acquisition times can be under one millisecond, making the channel acquisition delay acceptable in most applications. However in geographically distributed networks, the channel acquisition times can be tens of milliseconds. In applications in which sources send short transmissions, the overhead for acquiring a dedicated channel can be unacceptably long. For such applications, we would prefer a contention-based channel acquisition mechanism.

To add distributed contention resolution to a many-to-one virtual circuit, the senders delineate each packet, or burst of packets, with start and end cells. At each point where two or more branches of the virtual circuit come together, the switch tracks which of the incoming branches (if any) is currently propagating a burst through that merge point. If a start cell arrives on an incoming branch when some other branch is already propagating data, the switch rejects the new burst, optionally returning the start cell to the sender as a *reject cell* and discarding the cells received at that point. The reservation of the channel at the merge point ends when the end cell is received. To prevent the channel from becoming permanently unavailable due to a lost end cell, a timer deallocates the channel when no data has been forwarded within a fixed timeout interval. Switches also include trace information in start cells, allowing acknowledgements to be returned to senders on a reverse one-to-many path.

A single many-to-one channel with contention resolution can suffice when the average channel utilization is small, say less than 5%. For applications that send larger volumes of data however, collisions become too frequent to give acceptable performance. One way to address this is to simply use multiple many-to-one channels and have the senders randomly select a channel on which to send each burst. When there are n sources, k channels and the average number of busy sources is m, then the probability of collision is approximately

$$\sum_{i=1}^{n-1} \binom{n-1}{i} p^i (1-p)^{(n-1)-i} \left(1 - (1-1/k)^i\right)$$

where $p = m/n$ is the probability that any given source is sending a burst. When $n = 1000$ and $m = 1$, it takes 20 channels to produce a collision probability of 5% and 100 channels for a collision probability of 1%. In general, for n large and $n \gg m$, the collision probability is close to m/k, so it can take a large number of channels to keep the collision probability acceptably small.

To get better performance using local collision resolution, it's necessary to eliminate the fundamental inefficiency caused by the switches' inability to propagate more than one burst on a single channel. We propose the addition of a subchannel identifier to conventional virtual circuits, that switches along a many-to-one channel can re-map on a burst-by-burst basis in order to avoid collisions. The subchannel field is required in both the start and end cells, and the data cells of a burst, so it needs to be kept small so as not to take away too many bits from other parts of the ATM cell header. Fortunately, it does not take many subchannels to give vastly improved performance.

To implement subchannel remapping, switches monitor arriving data bursts as before, noting which incoming branches bursts come in on and what subchannel numbers they use on the incoming branch. Outgoing subchannels are assigned as needed on reception of start cells, and released on reception of end cells or on timer expiration. Data cells are forwarded using the subchannel number to identify the proper output subchannel, with the switch replacing the old subchannel number with the new one. If a start cell arrives on an idle input subchannel at a time when all the output subchannels have been assigned, the start cell is dropped or converted to a reject cell and returned to the sender, and the burst is discarded. As before, trace information is inserted into start cells to enable the return of acknowledgements. Note that the subchannel assignment mechanism is needed only for the downstream path, not for the acknowledgements.

If we have n sources, h subchannels and an average of m busy sources, the probability of a burst being blocked due to the unavailability of any subchannels is approximately

$$\sum_{i=h}^{n-1} \binom{n-1}{i} p^i (1-p)^{(n-1)-i}$$

When $n = 1000$, $h = 15$ and $m = 1$ the probability of a burst being discarded is less than 10^{-12}. If we increase m to 4, this becomes .00002. For $m = 8$, it is .02. To accommodate applications in which we expect more simultaneously active sources, we can use multiple virtual circuits with sources randomly selecting a virtual circuit on which to transmit. While the subchannel remapping takes place only within each virtual circuit, the number of virtual circuits needed to give low collision probabilities is now less than the average number of active sources, a big improvement over the version without remapping.

With 15 subchannels, a subchannel number can be encoded in four bits, making it possible to use the GFC field of the ATM cell header to carry the subchannel number (we leave one value unused, as an idle subchannel indication). At each merge point, a switch must store the status of all the outgoing subchannels. This status information includes the incoming subchannel number of the burst using the given output subchannel (if no incoming burst is currently using the output subchannel, the stored value is the idle subchannel value), the incoming branch that that the burst is coming in on and timing information used to determine when a given entry has been idle for more than the subchannel timeout period. In the recycling switch using binary branching, only one bit is needed to store the incoming branch information. If two bits of timing information are used, the switch must store seven bits per subchannel or 105 bits for 15 subchannels.

To make the reliable multicast mechanism useful in a general setting, it's important that there be some strategy for interoperability between switches that implement reliable multicast and those that don't. For the one-to-many mechanisms described in the previous section, interoperability is easy, since all that is required of switches that do not support the reliable multicast is that they be capable of forwarding the control cells and data on point-to-point virtual circuit segments. For the many-to-one case, the subchannel remapping mechanism can interfere with interoperability. For switches that propagate the GFC field without change along point-to-point virtual circuit segments, there is direct interoperability. However for switches that overwrite the GFC field (the usual case), some alternative approach is needed. The simplest way to address this is to allocate a block of consecutive virtual circuit identifiers on the link from a reliable multicast switch to a "standard" switch, and remap the virtual circuit subchannels to distinct virtual circuits. This is done by simply adding the subchannel number of forwarded cells to the first virtual circuit identifier in the allocated range. Similarly, when coming from a standard switch to a reliable multicast switch, distinct virtual circuits can be remapped to a single virtual circuit with subchannels. This mechanism for converting virtual circuit subchannels to consecutive virtual circuits is particularly useful in the context of delivery to end-systems, where the network interface circuitry will typically not be able to directly interpret the subchannel information. The use of distinct VCs at this point allows conventional network interface circuitry to properly demultiplex the arriving packets into separate buffers.

4. Many-to-Many Multicast Without a Relay

If one is not concerned with the provision of consistent delivery order to all receivers in a reliable multicast connection, it is possible to directly combine a many-to-one connection with a one-to-many connection, eliminating the intervening relay node. This requires extending the one-to-many mechanism to recognize different subchannels. The main cost of this extension is that for each of the 15 subchannels, the VXT entry must keep track of which transmission slot the last start cell on that subchannel specified (so that the data cells for the subchannel can be forwarded to the proper downstream branches). With a two byte transmission slot number, this overhead is 30 bytes, or less than two virtual circuit table entries in the WUGS switch.

Now, because this allows multiple sources to send packets directly into the one-to-many part of the connection, sources that are sending information concurrently must use distinct transmission slot numbers. The end systems can handle this in a variety of different ways. One is to simply assign slot numbers to sources statically. This is a reasonable approach when the number of senders is small. The other option is for the senders to allocate transmission slot numbers dynamically, using any general distributed resource allocation algorithm.

The elimination of the relay node makes it possible to structure a multicast connection that does not funnel through some common point and then fanout again to all the receivers. Instead we can have a more distributed situation in which data flows from senders through the multicast connection tree, branching at various points to reach the receivers. This can be implemented in the WUGS architecture by having each switch route the arriving cells into a many-to-one binary tree, followed by a one-to-many binary tree, allowing it to use the same mechanisms as it would if there were globally separate sections.

5. Implementation of Reliable Multicast in WUGS

This section details how the reliable multicast mechanism can be implemented in the Washington University Gigabit Switch (WUGS). This section assumes the reader is familiar with [16].

As discussed above, there are several cell types that make up the reliable multicast protocol: start cells, end cells, acknowledgement cells, negative acknowledgement cells, burst reject cells and data cells. The start, end, ack, nack and reject cells are all encoded using the ATM Resource Management cell type (PTI=110), with the first byte of the payload equal to the Reliable Multicast Protocol ID (123). The second byte of the payload contains a cell type field (0 for NOP, 1 for start, 2 for end, 3 for ack, 4 for nack and 5 for reject). The next two bytes contain the transmission slot number used for the one-to-many part of the protocol. The first bit of this two byte field is taken as the current color bit. The next eight bytes contain trace information, the subsequent eight bytes are reserved for future use, and the remainder of the payload is available for end-to-end information and is passed through by the switches without modification. In both data cells and control cells

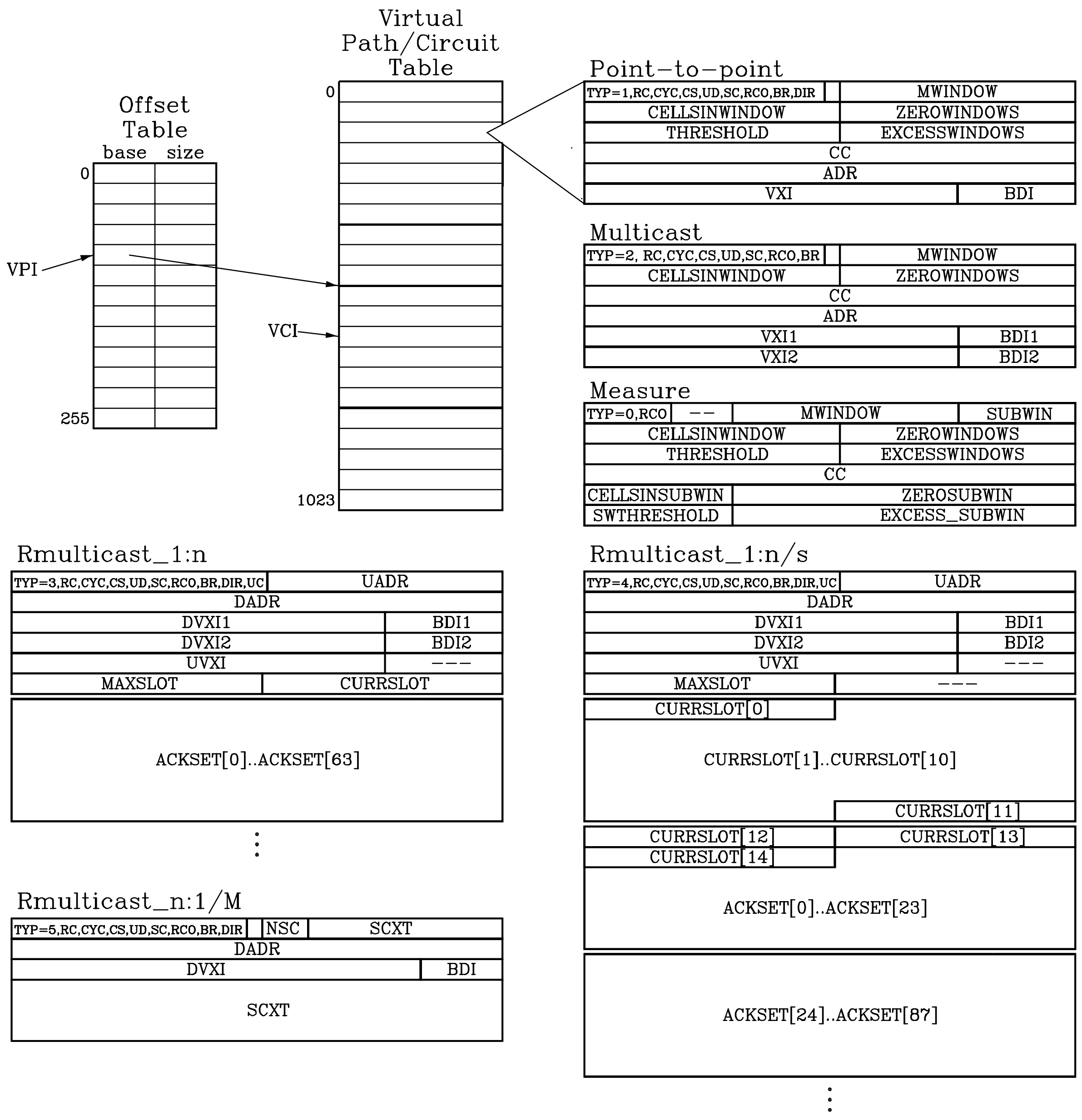


Figure 9: Virtual Path/Circuit Table Formats for Reliable Multicast

the GFC field of the standard ATM cell header is replaced by a subchannel field in which values 0–14 represent valid subchannels and 15 is reserved as an idle subchannel indicator. Switches implementing the reliable multicast protocol will operate on this field as discussed above and propagate it through to the outputs.

Figure 9 shows the format of the Virtual Path and Circuit Table used in WUGS. The VPI of an incoming cell is used to index an offset table that provides a base pointer and size parameter. For a non-terminating virtual path, the base pointer is zero, while for a terminating VP, the base pointer is > 0. For a non-terminating VP, the VPI is then used to index the main VP/VC table and the size paramenter is used to check that the given VPI is within the allowed range for non-terminating VPs (the given VPI must be strictly less than the size parameter). For a terminating VP, the required virtual circuit table entry is found by adding the base pointer value to the cell's VCI, after checking that the VCI is strictly less than the size parameter.

The entries in the VP/VC table can take one of several forms. The most common is the `point-to-point` form shown at the top right. This form is identified by a `TYP` field of 1 and is used to implement conventional point-to-point virtual paths and circuits. The first two bytes of the entry include the four bit `TYP` field plus several flag fields, most of which are defined as in the current WUGS system architecture document [16]. The Cell Counter (CC), Address field (ADR), Virtual Path/Circuit Identifier (VXI) and Block Discard Index (BDI) are also as in [16].

The Upstream Discard field (UD) is different than in [16]. The changes described here are needed when a switch is implementing reliable many-to-many multicast in which there is no global funneling through a common point. The UD field is a single bit, although the UD field in the internal cell format is still two bits. When a cell is received from the external link, the UD field of the internal cell is initialized, based on the value of the stored UD bit. When a cell is received at an IPP on the recycling path, the UD value in the cell is simply propagated. The initialization of the two bit UD field is done as follows: if the stored UD bit is 0, the field is set to 00, if the stored bit is 1 and the arriving cell is a start cell, then the UD field is set to 01, otherwise the UD field is set to 10. When the cell reaches an OPP where it is to be forwarded on the external link, the UD field and the STG field are evaluated. If the value of the cell's STG field is the same as the STG field in the OPP's maintenance register and the UD field is 10, the cell is discarded. If the STG fields match and the UD field is 01, the cell is forwarded to the recycling port, rather than the external link and the UD field is changed to 11. In all other cases, the cell is forwarded on the external link. When an IPP receives a cell on its recycling port with a UD field of 11 it performs the usual VXT lookup before forwarding it to the switch. It also examines the cell payload and if it determines that the cell is a reliable multicast cell (PTI=6 and Protocol ID=123) and that it is a start cell (the reliable multicast cell type field is 1), it changes the reliable multicast cell type field to 3 (making it an ack).

There are five new fields which allow more detailed traffic statistics to be gathered in hardware. These take advantage of the larger table entry that is necessitated by the reliable multicast protocol and will be discussed in more detail below. There is also a new one bit DIR field, which is inserted into all cells that pass through the table entry. This bit is used by the type 3, 4, 5, 6 and 7 entries as described belw.

In addition to the fields explicitly shown in the figure, every entry has a single `FIRST` bit, which is set for an entry that is the first of a multi-part entry and 0 for others. This is used to prevent cells with incorrect VPI/VCI values from accessing a multi-part VXT table entry incorrectly.

The next table entry format (`TYP=2`) is for (unreliable) multicast cells. This includes two VXI and BDI fields.

The next table entry format (TYP=0) is used to record statistics for a virtual path or circuit. This format is used for any unused virtual path or circuit. Also, by directing a copy of a given virtual path or circuit through a recycling port to such a table entry, traffic measurements for that connection can be taken. We take several different measurements. First, consecutive blocks of 2^{16} cell times (about 8.5 ms) are referred to as *measurement windows*. We count the number of windows in which no cells were received and the number of measurement windows in which at least a specified threshold were received. The MWINDOW field identifies the measurement window during which the most recent cell arrived (it is simply the value of the most significant 16 bits of the 32 bit clock register at the time the last cell was received). CELLSINWINDOW is a count of the total number of cells that have been received during that measurement window. The field EXCESSWINDOWS is incremented (modulo 2^{16}) every time the CELLSINWINDOW field is incremented to be equal to the value of the THRESHOLD field. When a new cell arrives, the number of windows in which no cells arrived can be easily determined by comparing the current measurement window number to the stored value of MWINDOW. The number of zero windows is then added to the ZEROWINDOWS field. The above measures are also taken for point-to-point table entries. For multicast entries, the EXCESSWINDOWS measure is omitted.

The measurement table entry also allows more fine-grained measurements. In particular we define consecutive blocks of 256 cell times to be *measurement subwindows* (this is about 34 μs). The combination of the MWINDOW and SUBWIN fields identifies a particular subwindow within the 2^{32} cell times that are tracked by the system's internal clock. The fields CELLSINSUBWIN, ZEROSUBWIN, SWTHRESHOLD and EXCESS_SUBWIN are used in the same way that the corresponding fields for the larger measurement windows are used.

There are six table entries specifically for reliable multicast. The first of these is labeled Rmulticast_1:n in the figure and has a type field of 3. This format is used for reliable one-to-many connections that are not "subchannel-aware." That is, they implement the algorithm described in section 2, including negative acknowledgements. The DADR and DVXI fields denote the *downstream* address and VXI fields. Similarly the UADR and UVXI fields denote the *upstream* address and VXI values. The CURRSLOT field holds the slot number associated with the most recent start cell that has been forwarded. The first bit of this field holds the current color bit, while the remaining 15 specify the transmission slot. The MAXSLOT field holds the maximum slot number for this virtual path or circuit. The ACKSET field is an array of three bit entries, each containing two bits of positive acknowledgement information and one NACK bit. The number of table entries used by a given reliable multicast connection is always at least two and can be larger, depending on the value of MAXSLOT. There is also a one bit DIR field, which is inserted into all cells that pass through the table entry. This entry also uses the DIR field in arriving ACK and NACK cells to determine which of the two children the cell came from and uses this information when updating the acknowledgement bits. This entry includes another one bit field called UC which is used to initialize the CYC field of acknowledgement (both positive and negative) that are forwarded through the entry. In particular, if the UC bit is 0, the CYC field is set to 00 and if UC is 1, the CYC field is set to 11.

The second type of table entry used for reliable multicast is the Rmulticast_1:n/s entry, which has a type field of 4. The /s stands for "subchannel-aware." These entries are used in much the same way as in the previous case, but are able to handle cells with different subchannel identifiers. The field `CURRSLOT[i]` contains the color bit and transmission slot number specified in the last start cell to be received with subchannel number `i`. When processing data cells with subchannel number `i`, the value of `CURRSLOT[i]` is looked up to determine which transmission slot the data cell belongs to. In other respects, the algorithm for processing these cells is just as described in section 2.

The third type of table entry used for reliable multicast is the `Rmulticast_n:1/M` entry, which has a type field of 5. Here, the $n : 1$ indicates that it is used in the many-to-one part of the reliable multicast protocol and the `/M` at the end means that this format is used at a merge point in the many-to-one connection. Only the downstream `ADR` and `VXI` fields are needed here. This entry also has a DIR field, which is inserted into cells passing through the entry, as already discussed. However, the use of the DIR bit of arriving cells is different for these entries. In particular, the DIR bit is used to determine which of two upstream neighbors the cell came from. This is used to access the subchannel mapping table. The stored DIR bit is also pushed into the trace field of the outgoing cell. The `NSC` field defines the number of output subchannels. This can be used to restrict the number of subchannels used to fewer than 15. The Block Discard Index placed in cells propagating through this table entry is computed by adding the BDI value stored in the table to the output subchannel number. This allows the block discard mechanism in the output port processor to separately track the start and end of packets on different subchannels, so it can perform packet-level discarding during output overload periods. The `SCXT` field contains the subchannel translation table. This is just a table of 15 entries, indexed by the downstream subchannel number. Each seven bit entry contains four bits specifying the number of the upstream subchannel using the entry (15 indicates that the output subchannel is not in use). Each entry also contains one bit indicating which of the incoming branches is using this output subchannel, and a two bit time value. These two timer bits are updated every time a cell is propagated on an active subchannel. The value assigned to these bits is taken from the switch's cell time clock. The particular bits that are stored are bits are programmable, allowing the time-out value to be selected. If one uses bits 20 and 21 (where bit 1 is the most significant bit), the value of these two bits change about every 250 ms. Whenever the `SCXT` is accessed, all of the stored timer bits are compared to the current value of the switch's corresponding clock bits. If the modulo 4 difference (current time minus the stored value) is greater than one, then no data has been passed on that subchannel for at least 250 milliseconds and the entry is made idle. An idle subchannel will thus normally become available to other sources within 500 milliseconds, even if the end cell has been lost. While a smaller timeout interval can certainly be used, there is little impact on performance, so long as applications send end cells at the end of a packet.

The type 5 table entry described above does not provide support for returning reject cells to sources when their start cells encounter all outgoing subchannels in use. While it is certainly possible to provide this feature, it does not appear to provide enough added value to warrant the added complication of including it.

The fourth reliable multicast VXT table entry format is identified by a `TYP` field of 6 and is referred to as the `Rmulticast_n:1/MC` format. Here the final `C` indicates that output conversion is implemented on this connection. That is, the outgoing VPI or VCI is replaced by the value stored in the table plus what would normally be the outgoing subchannel number in the cell. The subchannel number is omitted from the outgoing cell in this case. In all other respects it is the same as the `Multicast_n:1/M` format.

The fifth reliable multicast VXT table entry format is identified by a `TYP` field of 7 and is referred to as the `Rmulticast_n:1/FC` format, where FC stands for forward and convert. This is used at the input port of a switch in a many-to-one connection and converts a single virtual circuit to a subchannel. The format is almost the same as the point-to-point format. It adds a single four bit `SCH` field containing a subchannel number. This is simply inserted into the cell as it is forwarded. The field replaces the low order four bits of the `BDI` field in the point-to-point format. The remaining four bits of this field are not used.

The sixth reliable multicast VXT table entry format is identified by a `TYP` field of 8 and is referred to as the `Multicast_n:1/UR` format, where the UR stands for upstream return. This is used for returning acknowledgement cells back to a sender in a many-to-one connection. This format is almost identical to the `TYP 2` format. In order to add some additional flexibility, it uses the `DIR` bit in a different way than the other table entries. In particular, here the DIR bit is set to the upstream branch taken by the most recent ACK, NACK or NOP cell forwarded (these cells provide the branch indication in their trace field). Any other cell received on such a virtual circuit is forwarded along the branch indicated by the stored `DIR` bit.

6. Closing Remarks

In this report, we have not addressed issues of flow control and congestion control, in the reliable multicast context. For a one-to-many reliable multicast, one can use adaptive windowing techniques like those used in point-to-point protocols to adapt the sender's rate to accommodate slow receivers and/or congested links. Similar techniques are applicable to a many-to-one connection. The use of flow/congestion control on a one-to-many multicast does have the effect of slowing the connection data rate to that of the slowest receiver or most congested link. For distributed computing applications, this may well be the right thing to do. For information distribution applications, it's not clear that this is an appropriate approach. Indeed, there may be no single approach that is really suitable for all applications, as argued in [7].

We have neglected the problem of dynamically updating a reliable multicast connection in progress. Adding a new endpoint requires proper initialization of the acknowledgement state information for the new branch. Andy Fingerhut has developed a technique for doing this, which will be described in a forthcoming technical report. His approach is based on the observation that by temporarily disabling the forwarding of acknowledgements from a downstream neighbor, one can insert a new branch and initialize the acknowledgement state information without loss of consistency. The process is straightforward, although getting the details right is tricky. This approach can result in acknowledgements being lost, forcing "unnecessary" retransmissions, but this disruption can be kept quite small, since the time required to do the updating is fairly modest. If one must avoid loss of acks during connection updating, an alternative is to maintain two parallel connections, one for normal use and one for transitional purposes. When a new endpoint is to be added, it's first added to the second connection. After the new endpoint has been added to the second connection, the senders begins shifting transmission to the second connection. When there are no outstanding packets on the first connection, the owner signals to the network to add the new endpoint to that connection as well.

References

[1] Armstrong, S. A. Freier and K. Marzullo. "Multicast Transport Protocol," RFC 1301, 2/92.

[2] Bhagwat, P., P. Mishra and S. Tripathi. "Effect of Topology on Performance of Reliable Multicast Communication," *Proceedings of Infocom*, 6/94.

[3] Braudes, R and S. Zabele, "Requirements for Multicast Protocols," RFC 1458, 5/93.

[4] Chang, J. and N. Maxemchuk. "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, 8/84.

[5] Chaney, Tom, J. Andrew Fingerhut, Margaret Flucke and Jonathan S. Turner. "Design of a Gigabit ATM Switch," Washington University Computer Science Department, WUCS-96-07, 2/96.

[6] Crowcroft, J. and K. Paliwoda. "A Multicast Transport Protocol," *Proceedings ACM SIGCOMM*, 1988.

[7] Floyd, S., V. Jacobson, S. McCanne, L. Zhang, C. Liu. "A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," *Proceedings of ACM SIGCOMM*, 9/95.

[8] Holbrook, H., S. Singhal and D. Cheriton. "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation," *Proceedings of ACM SIGCOMM*, 9/95.

[9] Papadopoulos, Christos and Guru Parulkar. "Error Control for Continuous Media and Multipoint Applications," Washington University Computer Science Department, WUCS-95-35, 12/95.

[10] Paul, S., K. Sabnani, R. Buskens, S. Muhammad, J. Lin and S. Bhattacharyya. "RMTP: A Reliable Multicast Transport Protocol for High Speed Networks," *Proceedings of the IEEE Workshop on Computer Communications*, 9/95.

[11] Pingali, S., D. Towsley and J. Kurose. "A Comparison of Sender-initiated and Receiver-initiated Reliable Multicast Protocols," *Proceedings of SIGMETRICS*, 1994.

[12] Shacham, N. "The Design of a Heterogeneous Multicast System and its Implementation Over ATM," *Proceedings of the IEEE Workshop on Computer Communications*, 9/95.

[13] Whetten, B., S. Kaplan and T. Montgomery. "A High Performance Totally Ordered Multicast Protocol," *Proceedings of Infocom*, 1995.

[14] Turner, Jonathan S. "An Optimal Nonblocking Multicast Virtual Circuit Switch," *Proceedings of Infocom*, 6/94.

[15] Turner, Jonathan S. "Multicast Virtual Circuit Switch Using Cell Recycling," U.S. Patent #5,402,415, March 28, 1995.

[16] Turner, Jonathan S. and the ARL and ANG staff. "A Gigabit Local ATM Testbed for Multimedia Applications," Washington University Applied Research Lab ARL-WN-94-11.