

Multicast TCP via Concast Merged Acknowledgements

Billy Mullins, Jim Griffioen, Ken Calvert

[mullins,griff,calvert]@netlab.uky.edu

Department of Computer Science

University of Kentucky

Lexington, KY 40506-0046

859-257-6745

Abstract—One of the challenges in reliable multicast is approximating or replicating TCP’s congestion control algorithm. This has led to various proposals to adapt TCP itself for use over multicast. The drawback of these multicast TCP services is that they place significant processing load on the sender and they fail to deal with the problem of ACK implosion.

This paper shows how concast, a network-layer aggregation service, can be used in conjunction with a standard (unmodified) TCP implementation to support reliable multicast while avoiding ACK implosion and offloading ACK processing. Our prototype implementation shows that, even for relatively small groups, distributing the ACK processing and reducing ACK losses translates into significant improvements in end-to-end transmission rates.

I. INTRODUCTION

Building a reliable, flexible transport service on top of best-effort multicast has proven to be one of the most difficult problems in recent network history. The challenges arise from various aspects of the problem, including (i) the need to perform congestion control and loss recovery over multiple heterogeneous paths through the network; and (ii) sender-receiver processing asymmetries, including feedback implosion and processing overload at the multicast sender.

A wide variety of solution approaches have been explored. While some proposals largely ignore the problem of congestion control [1], others make it a central focus of the protocol. Solutions in the latter class often involve various ad hoc or “TCP-friendly” congestion control schemes [2], [3], [4], [5]. For these protocols, the asymmetry between sender and receivers poses a significant problem, which is typically addressed either by constructing an overlay of nodes with special functionality [4], [3], or by restoring symmetry and making all receivers be multicast senders [2]. Each of these poses challenges for deploy-ability.

An alternative approach that seems promising is to use TCP essentially unchanged, adding functionality *below* TCP in the protocol stack (in particular at the sender) to hide the existence of multiple receivers from its protocol mechanisms by preprocessing their acknowledgment messages. This approach changes TCP from a point-to-point service into a one-to-many service. A significant advantage of this approach is that it maintains the TCP abstraction for the application: sender and receivers interact with the service using the same basic set of calls as in a point-to-point

connection. Another advantage is that, from a congestion control perspective, the resulting channels behave *exactly* like a point-to-point TCP flow. Because they use the TCP congestion control mechanisms (not an approximation), no part of the multicast flow can obtain more than its fair share of the bandwidth.

In its pure form, this approach requires no special in-network processing. It has been investigated previously; an implementation known as single connection emulation (SCE) was created [6]. The main challenge in the absence of network support is the asymmetry: while the “shim” functionality hides the existence of multiple receivers from TCP, the shim itself must maintain and process (acknowledgment) state for each multicast receiver; clearly this places a significant processing load on the sender, and limits scalability. Moreover, as the number of receivers increases, so does the the potential for implosion and acknowledgment (ACK) loss near the source.

In this paper we investigate the use of a TCP-based approach with a modest amount of *general-purpose* network support, viz., the *concast* service. Concast [7] is a programmable network-level service that provides a *many-to-one* service abstraction. It presents a natural solution to many of the problems arising from sender-receiver asymmetry in large-scale multicast applications; as such, it has a variety of uses [8], [9], [10].

We call our service *Multicast Concast TCP (MCTCP)*. Our primary goal for MCTCP was to reduced the bandwidth and processing loads caused by ACK traffic converging at the sender. We also wanted to maintain backward compatibility so that MCTCP would function properly even if concast were only partially deployed. Finally, we wanted to maintain the standard TCP abstraction as much as possible, so that legacy applications could easily be converted to use MCTCP.

We begin with a brief overview of the concast service (Section II). Section III then introduces the MCTCP service abstraction. Section IV and following sections describe the general design of the MCTCP merge function and details of connection establishment, ACK processing, and option handling. Section VI describes our prototype implementation and presents performance results obtained from experiments run on our emulation network (EMULAB). Finally,

sections VII and VIII briefly describe how our work relates to past approaches, and offers some concluding remarks.

II. CONCAST OVERVIEW

Concast is a many-to-one communication service that provides the symmetric inverse of multicast: a group of senders transmit messages that are merged en-route to a common receiver R [10], [7]. As with multicast, an arbitrary number of group members (senders) are represented by a single group address G , hiding the group’s membership from the concast receiver R . Packets delivered to R are derived from the packets sent by members of G . A *concast flow* is uniquely identified by the pair (G, R) . Each flow is created by its receiver, and senders “join” the flow before they begin sending. The *Concast Signaling Protocol* (CSP) handles establishment of the relevant flow state in the network, i.e. at all concast-capable nodes on the paths from group members to the receiver. Each concast capable router maintains a *flow state block* that records a *merge specification* describing how packets are to be merged, and an *upstream neighbor list* (UNL) that records the next concast-capable nodes “upstream” (towards the senders) for this flow. The UNL is maintained using soft-state techniques similar to RSVP [11].

The concast service allows for different merging computations to be carried out by the network; the desired semantics (i.e. the merge specification) are supplied by the receiver at flow setup time. Merge specifications may be incrementally deployed throughout the network. At each concast enabled hop IP packets are checked for the Concast IP option. The Concast IP option contains the option flag, a concast sender id, and the group id. At each hop (G, R) is determined from the Concast IP option and the destination address. Packets are merged based on their respective (G, R) pair. The merge semantics are characterized by a *merge specification*, which defines (1) how datagrams delivered to the receiver are derived from datagrams transmitted by different senders (2) the timing of datagram forwarding and delivery; and (3) which datagrams are combined with each other (e.g. only packets containing the same sequence number are merged with each other). A custom merge specification can be specified by supplying definitions for the functions shown in Figure 5 which will be executed in the framework shown in Figure 1.

We have built a prototype concast implementation that comprises: a user-space CSP implementation; a user-space “merge daemon” framework for Java and one for TCL; and Linux kernel modifications to allow sender/receiver signaling via socket options, as well as intra-kernel routing of concast packets to and from the merge daemons. To increase merging performance, we have also developed a TCP merge daemon written in the “C” programming language. Unless otherwise noted, the “C” TCP merge daemon was used to collect performance statistics shown in this paper.

```

ProcessDataGram(Receiver  $R$ , Group  $G$ , IPDatagram  $m$ ) {
  FlowStateBlock fsb;
  DECTag t;
  MergeStateBlock s;

  fsb = LOOKUP_FLOW( $R$ ,  $G$ );
  if(fsb != NULL) {
    t = fsb.GetTag( $m$ );
    s = GET_MERGE_STATE(fsb, t);
    s = UPDATE_TTL( $s$ ,  $m$ );
    s = fsb.merge( $s$ ,  $m$ , fsb);
    if(fsb.done) {
      ( $s$ ,  $m$ ) = fsb.buildDatagram( $s$ );
      FORWARD_DG(fsb,  $s$ ,  $m$ );
    }
    PUT_MERGE_STATE(fsb,  $s$ , t);
  }
}

```

Fig. 1. Concast Framework Per-Packet Processing

getTag(m): a tag extraction function returning a hash or key identifying the message. Message m and m' are eligible for merging iff $\text{getTag}(m) = \text{getTag}(m')$.

merge(s , m , fsb): the function that combines messages together. The first parameter is the current merge state (i.e. information representing messages that have already been processed). The second parameter is the new message to merge into the saved state s . The third parameter is a “flow state block” containing information about the concast flow to which m belongs.

done(s): the forwarding predicate that checks s , the current message state, and decides whether a message should be constructed (by calling **buildMsg**) and forwarded to the receiver.

buildMsg(s): the message construction function, which takes the current message stats s , and returns the payload to be forwarded toward the receiver.

Fig. 2. Concast Merge Spec Functions

III. MCTCP SERVICE ABSTRACTION

One of our goals was to offer an abstraction similar to the standard TCP abstraction so that legacy applications could easily be converted to use MCTCP. Consequently, MCTCP offers a connection-oriented byte-stream service just like TCP. The primary difference is that data flows only in one direction (from the sender to the receivers). From the point of view of the applications, a multicast connection is established when the set of receivers invoke the *connect()* system call (at approximately the same time; see below) to connect to the sender. The sender accepts the connection requests from N clients using a single *accept()* call. Once the connection is established, data flows from the sender to receivers and TCP acknowledgements flow in the reverse direction.

Because MCTCP offers the same abstraction as TCP, MCTCP can be built on the standard TCP implementation without modification. Instead, multicast features are implemented in a concast merge specification and a system call that enables both multicast and concast on the socket. From the user’s perspective, establishing a reliable multicast TCP connection is identical to establishing a point-to-point connection except for one additional call to turn the connection

into a multicast connection, and one added call in the client to bind to a particular multicast address. Once data transfer begins, there is no difference; ACK merge processing is completely hidden from the user. The basic API calls are shown in Figure 3.

<u>Server</u>	<u>Client</u>
1. Open a (TCP) socket.	1. Open a (TCP) socket.
2. Enable MCTCP and join multicast group.	2. Enable MCTCP
3. Bind to the server port.	3. Bind to the multicast group/port
4. Call listen	4. Call connect()
5. Call accept to wait for a connection.	5. Receive data as normal
6. Send data as in traditional TCP	

Fig. 3. MCTCP API Call Sequence

Because the MCTCP abstraction allows the sending application to treat the receivers as if they were a single entity, it has the same nice scaling properties as the IP multicast abstraction. It should be noted, however, that throughput will be determined by the “slowest receiver”. Also, the probability of a packet being lost by some receiver grows exponentially as the number of receivers increases and can severely reduce the achievable throughput [12]. However, this is true of any reliable multicast scheme that does not support local retransmissions.

Given a kernel with TCP and concast support, we were able to implement our reliable multicast (MCTCP) abstraction simply by adding a new system call to enable MCTCP on a socket. This new system call performs three tasks. First, it joins the specified multicast group. Second, it creates a concast group and installs the merge specification needed for ACK aggregation. Third, it enables a kernel routine that converts multicast addresses to/from unicast addresses before the packet is handed to the TCP module. The conversion routine isolates the TCP module from the underlying multicast/concast services, creating the impression of the unicast channels TCP expects.

IV. MCTCP DESIGN

MCTCP’s basic proposition is that reliable multicast can be designed to avoid ACK implosion and minimize the sender’s ACK processing load by replacing TCP’s underlying unicast service with (1) a multicast service for data distribution in the forward direction, and (2) a concast service for ACK aggregation in the reverse direction. Designing a multicast service for the forward direction can be achieved in the obvious way: carry TCP packets in IP packets with a multicast destination address (as opposed to a unicast destination address). Designing an aggregation service for ACK traffic flowing in the opposite direction is not as easy. We address this problem in Section V.

Figure 4 illustrates the basic MCTCP architecture and initial connection setup phases. The initial step (not illustrated in Figure 4) is for the sender and receivers to join the appropriate multicast and concast groups. Like TCP, MCTCP then employs a three-way handshake to create the connection. In the first phase, each receiver sends

a concast SYN packet to the sender. To avoid maintaining TCP sequence number mappings at all intermediate routers, all receivers use the same starting sequence number. This can be accomplished in a variety of ways, but our current implementation uses a simple hash of the concast group address to determine the starting sequence number at MCTCP-enabling time. As the concast SYN packets travel toward the sender, they are merged together (see Section V) such that a single SYN packet representing all receivers is ultimately delivered to the sender.

Before delivering the SYN packet to the sender’s TCP module, the packet is passed through an address translation routine that changes the source address from a unicast address to a multicast address. This causes TCP to send the SYN+ACK response to the entire multicast group rather than the last concast router along the path. When the multicast SYN+ACK arrives at the receivers, it undergoes an opposite address translation in which the multicast destination is replaced with the receiver’s IP address. This translation ensures that the packet is associated with the correct TCP flow (i.e., TCP PCB). The final message of the three-way handshake is then sent by all receivers using concast to combine them into a single ACK delivered (after source address translation) to the sender’s TCP module.

From this point onward, the TCP module automatically uses the multicast group address as the destination address, receives combined (merged) ACKs representing the “slowest” of the receivers and updates the sliding window accordingly. At the end of the connection, a process analogous to the connection setup occurs, this time tearing down the connection and leaving the multicast and concast groups.

V. THE MERGE SPECIFICATION

The functionality needed to implement MCTCP without modifying TCP is largely contained in MCTCP’s concast merge specification.

As described earlier, the merge specification is defined in terms of four functions. Figure 5 outlines the functionality needed in each of the four functions in order to support MCTCP. Because the merge specification handles all TCP control packets (e.g., SYN, ACK, FIN) the operations performed by the *merge*, *done*, and *buildMsg* functions depend on the type of packet and the current connection state. The *getTag* function is used to identify the current connection state. Note that the TCP sequence number carried in the ACK packets does not change during the data transmission phase, because data only flows in the forward direction. As a result, the sequence number only changes when the session transitions from the connection setup phase to the data transmission phase, or from the data transmission phase to the connection teardown phase. Consequently, the *getTag* function simply needs to look at the sequence number to identify the current state of the connection. The following sections describe the merge spec behavior under each of the different connection states.

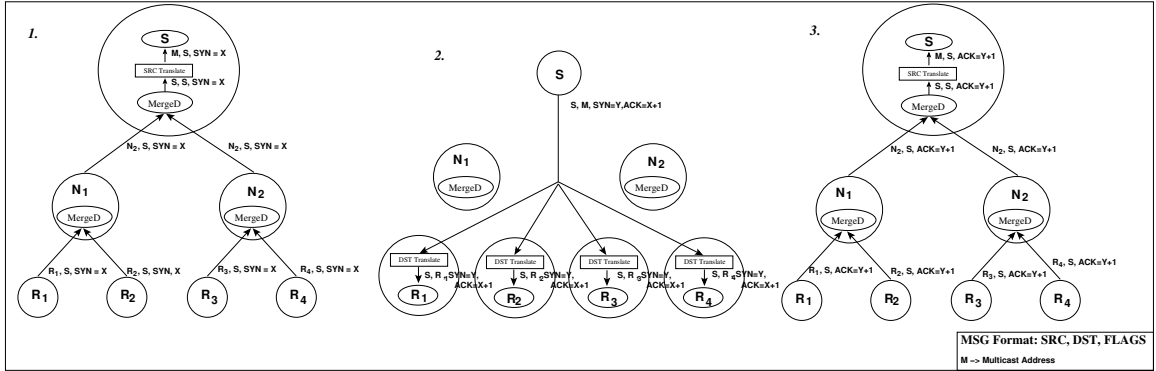


Fig. 4. MCTCP Connection Initialization

```

getTag(m):
    return tag = tcp_sequence_number
merge(s, m, f sb):
    if (connection_established) then
        if (not upstream_node_table{sender_ID}) then drop;
    else if (SYN) then
        upstream_sender_table.add(sender_ID);
    else drop;
    /* SYN or connection established */
    if (SYN) then "merge TCP options";
    window_table{sender_ID} = tcp_window_size;
    if after(tcp_ack_number, upstream_node_table{sender_ID}.ack_number) then
        upstream_node_table{sender_ID}.ack_number = tcp_ack_number;
    else retransmit = true;
done(s):
    if ((not connection_established) and (connection_establishment_timer = -1)) then
        connection_establishment_timer = ESTABLISHMENT_TIMEOUT;
        return false; /* not done */
    else if (not connection_established) then
        if (not SYN) then return false;
        if (connection_establishment_timer == 0) then /*timer has expired*/
            return true;
    if ((SYN) or (we have a new cumulative ack from all known senders)
    or (retransmit)) then
        return true; /*done, therefore call buildMsg*/
buildMsg(s):
    if ((not connection_established) and (SYN)) then connection_established = true;
    if (SYN) then "calculate tcp options";
    tcp_header.window_size = minimum_window_received(window_table);
    tcp_header.ack_number = minimum_ack_received(upstream_node_table);
    forward tcp control message;

```

Fig. 5. TCP merge specification pseudo-code using a simple timer based mechanism during connection establishment. Note: For simplicity, many error conditions and the connection termination pseudo-code have been omitted.

A. Merging During Connection Establishment

SYN merging can be accomplished in several different ways depending on how closely in time receivers are required to “connect” to the sender. In the least restrictive case, receivers join at any time. Although our current implementation does not support this, our decision to have receivers initiate the connection facilitates this model (but requires additional sequence number re-mapping on the receivers). Our current implementation assumes receivers join at roughly the same time—i.e., receivers must all connect within a specified time-frame (see Figure 5). Many connection synchronization approaches may be implemented, the desired approach simply needs to be encoded in the merge specification. Our current merge spec

contains both time, sender count, and a time/sender count combination mechanism for connection establishment. The method or combination of methods to be used can either be hard-coded into the merge specification or passed as a parameter from the concat receiver.

The merge daemons (“mergers”) change to the *connection established* state after they have forwarded the third message in the three-way handshake. At this point, the merge daemon has a list of its upstream children from which it expects to receive ACK messages (or possibly FIN messages).

B. Merging During Data Transmission

During the data transmission phase, TCP acknowledgement numbers for each receiver are maintained at their nearest downstream (to the concat flow) neighbor. Each merged maintains a list of its children’s (in the concat tree) IDs along with the highest cumulative acknowledgement seen from each child. When all children have acknowledged a byte(s), the merged constructs a representative cumulative acknowledgement and forwards it toward the multicast source.

C. Merging of TCP Options

Some TCP options are used quite frequently, usually to enhance performance in some way. To support these performance enhancements, we included support for a subset of the most common TCP options in our merge specification. In particular, we implemented the TCP window scale and the TCP maximum segment size(MSS) options. Both of these options are only carried on SYN packets, and have straightforward implementations. For the TCP MSS option we simply take the minimum MSS advertised by any receiver. In the case of the TCP window scale merging option, we set all clients to use the same window scale. Our goal was to simply allow the window scale option to function with our implementation. If the clients do not cooperate on selecting a window scale, the result is that a smaller advertised window is recognized, and as a result for high-bandwidth large-delay connections a performance loss can result.

Other TCP options are more difficult to support. For example, the TCP timestamp option is difficult to implement since there is no simple way to collect a “global” timestamp from all the competing MCTCP clients. As a consequence, the timestamp-supported Protection Against Wrapped Sequence (PAWS) feature cannot be easily supported.

VI. RESULTS

To evaluate the impact concast ACK merging has on multicast throughput, we implemented a prototype system in the Linux kernel and measured performance using an emulated network topology. In particular, we recorded the sender’s processing load with and without MCTCP to quantify the benefits of distributing the ACK processing. We also measured the overall throughput achievable with and without MCTCP.

Figure 6 shows the network topology used in our tests. The tests were run on the Kentucky EMULAB facility (based on software from the University of Utah [13]). Each node in the topology was a 1.5Ghz Pentium with multiple 100 Mbps network interfaces. The topology consisted of one sender, 18 multicast receivers ($R_0 - R_{17}$), and three levels of merge daemons ($M_0 - M_8$). To evaluate performance with and without merge processing, we ran the system with merging only at the sender (like SCE [6]), and with merging at the sender and all intermediate nodes (MCTCP). To measure the benefits when concast is only partially deployed, we also measured performance with merging at the sender and a subset of the intermediate routers.

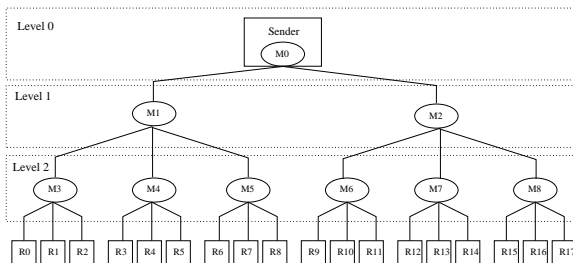


Fig. 6. Experimental Topology (EMULAB)

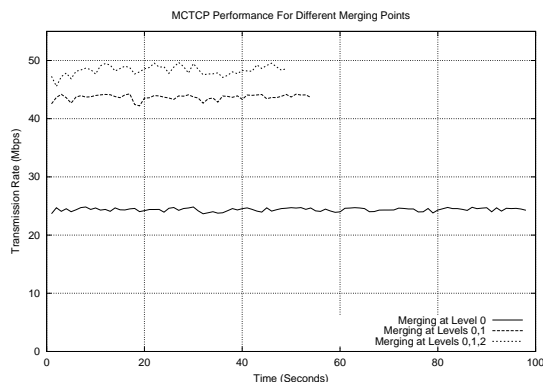


Fig. 7. Multicast Throughput Results

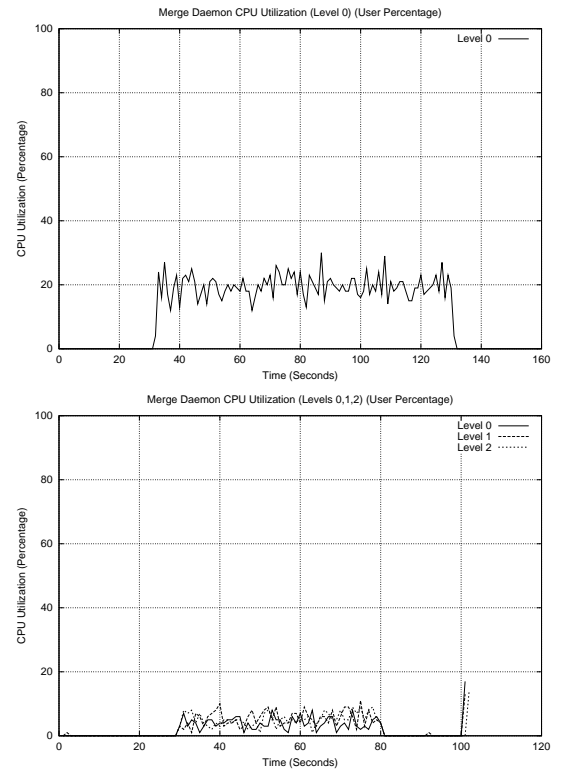


Fig. 8. Total CPU Utilization for sender-only merging (Level 0) and MCTCP (Level 0-2).

Figure 7 illustrates the throughput achievable when merging is enabled at various levels of the multicast tree, ranging from the sender-only (Level 0) to all routers (Levels 0,1,2). The graph shows that even with this rather small topology, throughput can be significantly improved (roughly a factor of 2) by pushing ACK merging/aggregation into the network. Simply offloading some ACK processing from the sender to Level 1 nodes resulted in a significant throughput improvement.

Figure 8 shows the CPU utilization at the merging nodes at each level of the hierarchy. The graphs show only the user-level load (i.e., the load resulting from merged processing).

Figure 8a shows the CPU load (at the sender) with sender-only merging. Because the sender was responsible for merging all ACK packets, the queues became longer, lengthening the time before the window could be advance and slowing the overall throughput. It should be noted, that if we combined system and user-level CPU utilization for the sender-only case the result was 100% CPU utilization at the server node.

Figure 8b shows the average CPU load at merged’s from each level of the MCTCP implementation. It shows the load is evenly distributed across the merging levels and is only a small fraction of the load experienced by the sender in the sender-only case. Even when including the system-level CPU utilization, the maximum load was less that 70% for all merge nodes. The end result of MCTCP is a significant improvement in end-to-end throughput.

VII. RELATED WORK

Reliable multicast has been widely studied in the past using approaches that range from network level services, to active network services, to application level services [3], [4], [14], [15], [16], [17], [1], [18], [2], [19], [5]. The most closely related work is the Single Connection Emulation (SCE) approach [6]. Like our work, SCE replaced the underlying unicast transport service with a multicast service. Unlike our work, SCE relied on unicast channels in the reverse direction with all merge processing located at the sender. As we have shown, the processing load imposed on the sender by such an approach can significantly reduce or limit the achievable throughput.

Another difference between our approach and SCE is the fact that connections originate at the sender in SCE whereas they originate at the receiver in MCTCP. Because connections originate at the receiver in MCTCP, receivers can join/leave at any time.

VIII. CONCLUSIONS

Many reliable multicast protocols have proposed ways of approximating, or replicating, TCP's congestion control algorithm. The drawback of these multicast TCP services is that they place significant processing load on the sender and they ignore the problem of ACK implosion.

In this paper, we described a new reliable multicast protocol, MCTCP, that avoids the problem of ACK implosion and offloads ACK processing to routers in the network. Performance results from our Linux prototype demonstrate that significant performance gains can be achieved by offloading ACK processing into the network. MCTCP maintains the TCP service model abstraction so legacy applications can be easily converted to use the new service and it can be implemented without any modifications to the kernel TCP modules. The model also allows for receivers to join/leave the connection dynamically.

REFERENCES

- [1] T. Speakman, D. Farinacci, S. Lin, and A. Tweedly, "The PGM Reliable Transport Protocol," August 1998, RFC (draft-speakman-pgm-spec-02.txt).
- [2] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," in *SIGCOMM*, Cambridge, Massachusetts, September 1995.
- [3] R. Yavatkar, J. Griffioen, and M. Sudan, "A Reliable Dissemination Protocol for Interactive Collaborative Applications," in *The Proceedings of the ACM Multimedia '95 Conference*, November 1995, pp. 333–344.
- [4] S. Paul, K. Sabnani, J. Lin, and S. Bhattacharyya, "Reliable Multicast Transport Protocol (RMTP)," *The IEEE Journal on Selected Areas of Communication*, 1996, (see also the Proceedings of IEEE INFOCOM'96).
- [5] Katia Obraczka, "Multicast Transport Protocols: A Survey and Taxonomy," Tech. Rep., University of Southern California Information Sciences Institute, November 1997.
- [6] R. Talpade and M. H. Ammar, "Single connection emulation: An architecture for providing a reliable multicast transport service," in *Proceedings of the 15th IEEE Intl Conf on Distributed Computing Systems*, June 1995.
- [7] Kenneth L. Calvert, James Griffioen, Billy Mullins, Amit Sehgal, and Su Wen, "Concast: Design and implementation of an active network service," *IEEE Journal on Selected Area in Communications (JSAC)*, vol. 19, no. 3, pp. 426–438, March 2001.
- [8] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen, "Concast: Design and Implementation of a New Network Service," in *Proceedings of International Conference on Network Protocols*, November 1999.
- [9] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen, "Building a programmable multiplexing service using concast," in *Proceedings of International Conference on Network Protocols*, November 2000.
- [10] Ken Calvert, Jim Griffioen, Billy Mullins, Amit Sehgal, and Su Wen, "Implementing a Concast Service," in *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.
- [11] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network*, September 1993.
- [12] Supratik Bhattacharyya, Donald F. Towsley, and James F. Kurose, "The loss path multiplicity problem in multicast congestion control," in *INFOCOM (2)*, 1999, pp. 856–863.
- [13] "Netbed: Network emulation facility," <http://www.uky.emulab.net>.
- [14] Injong Rhee, Nallathambi Ballaguru, and George N. Rouskas, "MTCP: Scalable TCP-like congestion control for reliable multicast," in *INFOCOM*. IEEE, 1999.
- [15] L. Lehman, S. Garland, and D. Tennenhouse, "Active Reliable Multicast," in *Proceedings of the INFOCOM Conference*, March 1998.
- [16] Sneha Kumar Kasera, Supratik Bhattacharyya, Mark Keaton, Diane Kiwior, Jim Kurose, Don Towsley, and Steve Zabele, "Scalable Fair Reliable Multicast Using Active Services," *IEEE Network Magazine*, February 2000.
- [17] Christos Papadopoulos, Guru Parulkar, and George Varghese, "An Error Control Scheme for Large-Scale Multicast Applications," in *Proceedings of the INFOCOM '98 Conference*, 1998.
- [18] B. Cain and D. Towsley, "Generic Multicast Transport Services: Router Support for Multicast Applications," Tech. Rep. CMPSCI TR 99-74, Umass, October 1999.
- [19] H.W. Holbrook, S.K. Singhal, and D.R. Cheriton, "Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation," in *Proceeding of the ACM SIGCOMM'95 Conference*, November 1995.