Leveraging emerging network services to scale multimedia applications

K. Calvert, J. Griffioen, B. Mullins, S. Natarajan, L. Poutievski, A. Sehgal, and S. Wen

©University of Kentucky, Dept. Of Computer Science.
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046

SUMMARY

Multicast services have been used to transmit multimedia data to large receiver groups. Only recently have counterpart network services been introduced that provide similar scalability and anonymity in the opposite direction (i.e. messages from a group of senders destined for a common receiver).

In this paper, we explore how these many-to-one services, specifically the *concast service*, can be used to improve the scalability and performance of multimedia applications. In particular, we show how such services can be used in both the control and data planes to overcome well-known scalability problems (e.g. with RTP) that are difficult to solve effectively with end-system approaches alone. We validate our solutions by presenting experimental results taken from prototype video and audio applications we designed and implemented. Our initial results show reductions of as much as two orders of magnitude in packet loss rates using these generic services.

1. Introduction

Multicast services provide data delivery to a group of receivers, and allow applications to scale up in group size by providing both *bandwidth savings* (duplicating packets in the network instead of transmitting duplication at the end systems), and *anonymity* (hiding group membership information from senders and allowing them to deal with the group as a single entity). Multimedia applications have

Contract/grant sponsor: Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Material Command, USAF, under agreement number F30602-99-1-0514, and NSF CISE RI Grant number EIA-0101242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, the Air Force Research Laboratory, or the U.S. Government.

long been the primary users of multicast services. For example, real-time audio and video applications have been running over the Internet for years now.

Traditionally, the only way for a multicast application to send information in the reverse direction (i.e. from many group members to a single destination) is via unicast (or possibly multicast) flows from the group members to the single destination. For example, many deployed multimedia applications need control information (feedback) to be sent from the receivers to the source; these applications either use UDP or require all receivers to also act as multicast senders. This limits the scalability of such applications, because (i) as data from many sources is "funneled" toward one receiver, the load may exceed available resources (either in the network or in the end system), resulting in *implosion* [18, 19]; and (ii) the receiver is forced to deal with the many group members individually, destroying anonymity.

These limits are difficult to overcome with end-system-only mechanisms. Existing solution approaches typically fall into one of two categories. Either the end-systems learn/guess/infer the group size and adjust the feedback rate appropriately [19, 20, 5], or they create an overlay network in the reverse direction to aggregate control information [23, 17].

Recently, however, various network-level mechanisms have been proposed to aid in overcoming these limits. For example, Pragmatic General Multicast [22] is a protocol based on a set of router-based mechanisms that can improve the scalability of multicast applications that use sender-based retransmissions. Follow-on work, namely Generic Router Assist [9], comprises simple router-based mechanisms that can be used in a variety of ways to improve scalability and performance of group applications. Yet another service, Concast [7, 8], is a backward-compatible many-to-one service that allows application-specified *merging* functionality to be loaded into supporting routers, thereby providing the same kind of benefits as multicast. Concast changes the limiting factor on scalability from the group size to the branching factor at any node, thus providing an exponential increase in scalability.

In this paper, we show how these new network-based mechanisms (specifically the concast service) can improve the performance and scalability of multimedia applications in two ways. First, in the control plane, by permitting feedback to be transmitted to the source without implosion and with anonymity, and second, in the data plane, by allowing multimedia streams to be combined *en route* to a point of convergence. The latter capability is an alternative to the placement of transcoding or combining servers throughout the network [4, 3, 13]. Placing such servers present some practical difficulties: identifying the "correct" location for a server is difficult, and moreover it may vary with group membership and network congestion. However, using a generic network-level service like *concast*, transcoding and merging of data flows can be done without any a priori knowledge of where the transcoding takes place. This approach implies the use of network computation resources on behalf of applications; in this paper we also investigate the computing load imposed by this kind of application.

The remainder of this paper is organized as follows. Section 2 gives a detailed description of our implementation of the concast service on which we have developed solutions to the problems described earlier. Section 3 describes an end-to-end service that uses concast to aggregate Real-Time Control Protocol (RTCP) feedback for media applications. Section 4 describes two example end-system combining services, one for video, the other for audio. We describe a prototype implementation of the video and audio service and present experimental results that demonstrate the scalability of our approach with as much as two orders of magnitude reduction in packet loss rates using our video service. Section 5 discusses related works, and Section 6 summarizes our conclusions.

2. Concast Service and API

Concast is a many-to-one communication service that provides the symmetric inverse of multicast: a group of senders transmit messages that are merged enroute to a common receiver R. 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 service is flexible in that it 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. We have built a prototype concast implementation which comprises three major components, namely the *Concast Signaling Protocol* (CSP), *Merge Daemons* (MERGEds), and the *Concast API*. The following sections describe each of these three components of the service.

2.1. Concast Signaling

The *Concast Signaling Protocol* (CSP) establishes concast-related state in network nodes. The essential functionality entrusted to the protocol is to "pull" the *merge specification* specified by the receiver to the nodes on the path from the the senders to the receiver.

Each concast capable node maintains a *Flow State Block* (FSB) for each active concast flow. The FSB contains the *merge specification*, *state information* for any in-process merges, and an *Upstream Neighbor List* (UNL) that contains the addresses of the node's "children" in the concast tree.

A network node (end systems or routers) can support concast at one of three levels. The lowest level is *no support*. These are legacy nodes, unaware of concast, that simply forward concast packets. The highest level is *full concast support*. Nodes at this level support the merging framework, the creation of new flows and the joining of existing flows. Hosts at this level are capable of supporting both sender and receiver applications. The middle level of support is *partial support*. Nodes at this level do not implement the merging framework, and thus are capable of supporting concast senders, but not merging

The merge specification is only communicated between full support nodes. Thereby reducing the security risk that untrusted nodes can pose if they are allowed to merge data.

The CSP protocol is implemented using two distinct daemons. Creation and merging of flows is supported by an *RCSPd* (i.e. the *R*eceiver capable *C*oncast *S*ignaling *P*rotocol *d*aemon) while joining, but not merging is supported by the *SCSPd* (i.e. the *S*ender capable *C*oncast *S*ignaling *P*rotocol *d*aemon). Nodes that provide partial support run only the SCSPd while nodes that provide full support run both the RCSPd and the SCSPd [†].

All CSP messages are sent as regular IP unicast datagrams with the (protocol) type field set to CSP. Messages traveling downstream (i.e. towards the receiver) carry the Router Alert IP option. This

[*] Note that for concast, both the receiver and group members signal before sending; for multicast, both are also required to signal, but the two sets coincide.

[†] The purpose of the separation is to provide a subset of functionality to machines only requiring concast send functionality, and to establish a merge specification trust boundary.

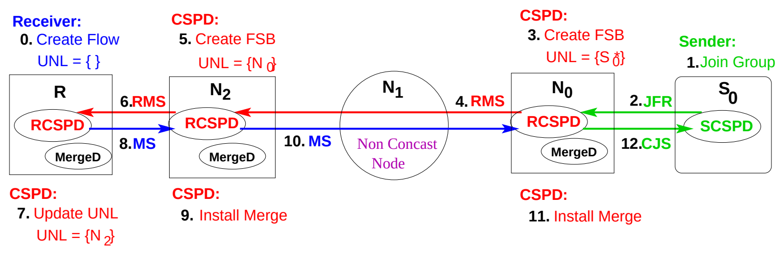


Figure 1. Concast Signaling Events

enables all concast capable nodes on the path towards the receiver to intercept and process CSP packets. The source address of CSP messages is reset at each hop to be the unicast address of the last node to process the packet. This is crucial because it enables connectivity problems in the concast tree to be detected via ICMP, and is used to build the UNL.

Figure 1 depicts an example sequence of signaling events. Nodes N_0, N_2 and R have full concast support, node S_0 has partial support, and node N_1 has no concast support. Concast processing begins with the receiver application on R signaling to the local RCSPd its desire to receive concast messages from a specific concast group G, using the supplied merge specification. The RCSPd creates an FSB for the flow G, with an empty UNL.

Before an application can send to R, it must join group G. It signals to the local SCSPd on S_0 which creates a FSB for the sender and sends a *Join Flow Request* message (JFR) message with source = S_0, destination = R with G as the group id. The JCF message travels towards R until it encounters a concast-capable router with full support. The node, N_0 in this case, intercepts the CSP packet and passes it to the local RCSPd. If a FSB for G already exists at N_0, the RCSPd simply adds S_0 to the UNL of the flow and sends a *Concast Join Success* (CJS) message back to S_0, If no FSB is not found, one is created and the UNL is initialized to $\{S_0^*\}$ where '$*$' indicates that the upstream member is a node with partial support. N_0 then sends a *Request for Merge Specification* (RMS) message with source = N_0, destination = R, containing G. The RMS message travels towards R, with a new FSB being created at each concast-capable hop until it reaches a node that already has a FSB for the flow. Note that node N_1 has no concast support and simply forwards the concast messages through. When the RMS message reaches N_2 , node N_0, the source of the RMS message, is added to N_2's UNL. At R, N_2 is added to R's UNL list. Finally, a *Merge Specification* (MS) message containing the code for the merge specification is sent to all upstream nodes, in this case with source = R and destination = N_2. When N_2 receives the MS message, it stores the merge specification in its FSB and forwards the MS message (with source = N_2) to all members of its UNL who are not marked sender-only. A CJS message, not containing the merge spec, is sent to those marked sender-only. Next, if there exist two or more upstream neighbors, then a MERGEd is spawned. The node functions in the merging mode, passing all packets of the flow to the MERGEd. If only one upstream neighbor exists then the node functions in a forwarding mode, transparently forwarding all concast packets without going through the MERGEd

In summary, an RMS message propagates through the network setting up flow state until it reaches a node that has the merge specification, which then replies with a MS message that propagates back upstream towards the sender via the Upstream Neighbor Lists. Thus new senders connect to the concast tree where their path (towards the receiver) intersects the existing tree.

When a sender wishes to leave the flow it signals to the local SCSPd which sends a *Leave Flow Request* (LFR) message towards R. N_0 processes this message by deleting S_0 from its UNL. If the resulting UNL is empty then N_0 turns around and sends a LFR message towards R. Normal operation continues if the list is not empty. No message is sent in response to the LFR message. When the receiver wishes to terminate listening on the flow it signals the local RCSPd. The termination of the receiver is equivalent to the termination of the flow, hence all state in the network needs to be purged. The RCSPd on R thus sends a "No Such Flow" error message to all its upstream neighbors. All receiving RCSPds terminate the MERGEd for the flow and propagate the message to their upstream neighbors and purge the state. All SCSPds report the error to the sending applications and then purge the state related with the flow.

2.2. The Merge Daemon Framework

The *Merge Daemon* at concast-capable routers must recognize and intercept concast datagrams. Fortunately this is something concast has in common with other protocols, such as RSVP, which also need hop-by-hop processing. The *Router Alert* IP option [12] is defined for the specific purpose of flagging datagrams for closer examination. The presence of Router Alert in the IP header causes a router, during forwarding, to check for a particular payload type (e.g., RSVP). Similarly, packets associated with a concast flow have a *Concast* IP option. Concast processing is triggered by the presence of the concast group id G contained within the *Concast* IP option. The actual path between two concast-capable nodes may include multiple concast-oblivious hops which treat concast packets as regular IP datagrams. When a datagram is diverted for concast processing, the flow state block for the G flow, if any, is retrieved. If there is no FSB for the flow, the packet is silently dropped. Otherwise, the packet is handed off for merge processing, according to the merge specification.

In the context of a particular concast flow, we define a *datagram equivalence class* (DEC) as a set of packets to be merged together. Packets from multiple DECs may be present in the network at the same time. Two packets are in the same DEC if they have the same hash or fingerprint. Because packets are processed one at a time as they arrive, each merge daemon maintains a *merge state block* for each active DEC of a flow. To limit the amount of per-flow state, the size and number of the active merge state blocks is limited by the merge processing framework. When merge processing is "finished", a concast datagram is constructed using information in the merge state block, and forwarded to R. To prevent abuses, the merge processing framework will *only* emit packets that belong to the flow G.

From the foregoing discussion, it should be clear that the merge specification must include the following information:

- A mapping from concast datagrams to datagram equivalence classes.
- A description of the merge state block.
- A description of the "merge" computation, in a form that maps an arriving message and a merge state block into a new merge state block.
- A description of the conditions under which merging is considered "finished".

- A description of the forwarded messages, in the form of a mapping from merge state to concast datagram payload.

The receiver is responsible for supplying these descriptions at flow creation time. The most general approach (and the approach we have implemented) is to supply high-level machine independent code (e.g. Java) that defines the following:

type `MergeState`: The type of the state maintained for each message equivalence class.

function *getTag*(m:`AppMessage`) **returns** `AppTag`: A *tag extraction* function returning a "key" identifying the DEC to which a packet belongs. Messages m and m' are in the same DEC iff *getTag*(m) = *getTag*(m'). The processing framework allows instances of type `MergeState` to be stored and retrieved using these keys.

function *merge*(s:`MergeState`, m:`AppMessage`, f:`FlowStateBlock`) **returns** `MergeState`: The merge computation itself. The first parameter refers to the `MergeState` for the DEC to which the incoming message (parameter m) belongs. The third parameter is the Flow State Block for the concast flow to which m belongs. The Flow State Block is well-known and is accessed read-only. It contains information, such as the interface to the Upstream Neighbor List, which may be needed by the merge computation.

function *done*(s:`MergeState`) **returns** `boolean`: The *forwarding predicate* that decides whether a message should be constructed (by calling *buildMsg*) and forwarded to the receiver. This function may set up timeouts to arrange for later processing in case additional packets do not arrive.

function *buildMsg*(s:`MergeState`) **returns** `AppMessage`: The *message construction* function, which takes the current merge state and returns a concast packet to be forwarded toward the receiver.

Pseudo-code for the generic hop-by-hop processing performed by the merged is shown in Figure 2. The functions `lookUpFlow()`, `findMergeState()`, and `saveMergeState()` are system functions that have the expected semantics.

Our implementation also allows the *merge* function to arrange for the last part of the processing loop (i.e. the *done* function and subsequent construction and forwarding of a packet) to be invoked with a particular merge state block after a timeout period. This makes it possible to ensure that a message is forwarded at some point. A flow is allowed to have at most one pending timeout associated with each active merge state block.

The merge processing framework needs to protect the integrity of the node while carrying out user-specified merge computations. For example, any invocation of a user-supplied method should be forcibly terminated if it does not complete within a time limit. One way to achieve this is by making the language in which the methods are encoded very restrictive, for example permitting only straight-line programs. Alternatively, the specification might be given in a more powerful language and checked at flow creation time for forbidden constructs such as loops, invocation of methods other than those explicitly permitted, etc. Our current MERGEd implementation allows for merge specifications to be written in Java. A base Java framework has been designed that when executed will load copies of the merge specification, using in-band signaling or HTTP Get requests.

```
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);
    }
}
```

Figure 2. Concast Framework Per-Packet Processing

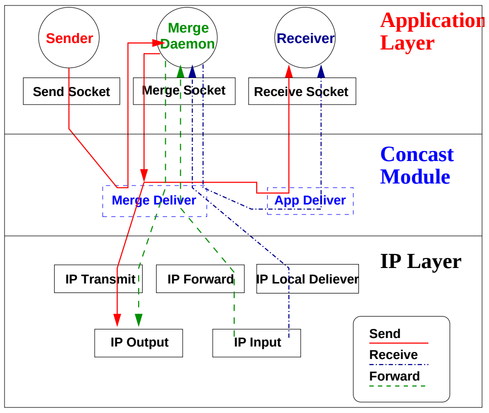


Figure 3. Concast intra-kernel routing

2.3. The Concast API

The concast API provides the interface by which applications join/leave concast groups and send/receive concast datagrams. The API is implemented as a loadable kernel module under Linux.

```
setsockopt ( sock, IPPROTO_IP, IP_CONCAST_CREATE, &mergespec, sizeof(mergespec) )
setsockopt ( sock, IPPROTO_IP, IP_CONCAST_JOIN, &flowspec, sizeof(flowspec) )
```

```
                                        struct  mergespec {
   struct flowspec {                      struct flowspec fs;
     ulong ReceiverIPAddress;             uint   mspec_length;
     ulong AppID;                         char   *mspec;
   }                                    }
```

Figure 4. Synopsis of the concast user API socket options.

The module is kernel version portable, and uses the Linux NetFilter API [1]. The Linux NetFilter API provides a method of adding "on-the-fly" IP hooks and socket options to an already executing Linux Kernel. The concast module provides intra-kernel routing of concast datagrams between the different kernel components which can be seen in Figure 3.

Applications join a concast flow by creating sockets in the normal fashion, and then associating the socket with a concast flow. Two socket options are provided to associate a user socket with a concast flow. A synopsis of these concast socket options and their use is shown in figure Figure 4. The IP_CONCAST_CREATE call is invoked by the receiver to initiate a concast flow. The create call requires a flow specification and a merge specification that are loaded into the local RCSPd. The IP_CONCAST_JOIN call takes a flow specification, and is used by senders to join the concast group. The join call triggers the SCSPd to begin the CSP join process.

Having defined the three main components of the concast service and their implementation, we now turn our attention to ways in which the concast service can be used to simplify and/or improve the performance of multimedia applications. The following sections describe example uses and provide experimental results taken from our prototype system.

3. Scaling RTP/RTCP

The *Real Time Protocol* (RTP) [21] is a general purpose protocol that has been used by a wide range of real-time video and audio applications. Because real-time applications often require feedback from the participants in the multicast group, RTP's control protocol, RTCP, provides a generic feedback mechanism called Receiver Report (RR) messages. RR messages provide the source with information about the performance of each receiver in the group. For example, given information about the loss rates at receivers, the source can adjust its transmission rate, encoding scheme, number of multicast groups, etc. The RR message format is shown in Figure 5. Each RR corresponds to a single reporting interval and carries information about the source of the report message, the current loss rate, cumulative losses, how much of the stream has been received (the highest numbered packet seen), inter-packet delays (jitter), and information identifying the reporting interval. Each RR may contain reports from multiple sources.

Each receiver in the group periodically transmits an RR describing its current performance information. As the group size increases, the number of RRs transmitted grows and implosion becomes

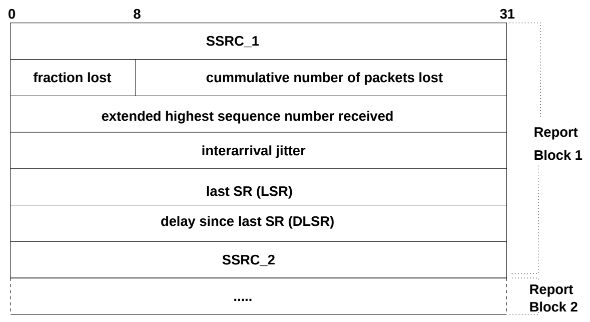

0
8
31
SSRC_1
fraction lost
cummulative number of packets lost
extended highest sequence number received
interarrival jitter
last SR (LSR)
delay since last SR (DLSR)
SSRC_2
.....
Report Block 1
Report Block 2

Figure 5. Format of Report Blocks in the Receiver Report

a problem. To prevent implosion, RTCP uses information about the current group size to determine the rate at which each receiver transmits RRs: as the size of the group increases, the rate of RR transmission decreases. In order to discover the group size, all RR's are multicast to all members of the group. As a result, all receivers see all RR's from all machines (and learn the group size).

Although this approach ameliorates the problem of implosion at the multicast source, it has several drawbacks. Because RRs are multicast, the anonymity of the multicast abstraction (i.e., group members are not known) is destroyed. Also, multicasting RRs means that all group members become vulnerable to implosion (not only the multicast source). Increasing the time between RRs as the group size grows also increases the time it takes for the application to react to changes in network conditions. Because of the difficulty and importance of this problem, various researchers have proposed end-system approaches to estimate the optimal reporting interval for RRs [19, 20].

Using concast for RR transmission has several obvious advantages. The main advantage is that feedback messages only need to be sent (via concast) to the real-time data source. Other nodes can remain oblivious to the size and membership of the group. Group membership information can also feedback at a constant rate, relying on the concast service to limit the bandwidth used by feedback messages.

3.1. Merging RTCP Receiver Reports

In the following we present three possible concast merge specifications that can be used to efficiently combine RR reports so that implosion is avoided. It is important to note that the RTP/RTCP protocol does not specify *how* the application uses the information from RRs. Typically the application is only interested in *summary* information, such as the maximum loss rate, minimum delay, minimum sequence number received, average jitter, or longest reporting interval. However, in certain cases the application may want to see each report as opposed to summary information (e.g. to learn group membership); in these cases the use of concast can still be advantageous, to prevent implosion.

In view of the application-dependent semantics of RR messages, we present merge-specifications for three classes of applications: (1) those that want *summary* information, (2) those that want *complete* information, and (3) applications that use customized RR packets. To simplify the presentation of the three classes of merge specifications, we will use the packet field names listed in Figure 5.

3.2. Aggregating Receiver Reports

Generally a multicast sender is not interested in each receiver's individual report, but rather wants to know information such as the maximum loss rate, the minimum delay, or the minimum sequence number received. Such information can be obtained by deploying a merge specification that invokes the desired operator (e.g., `min`, or `max`) on incoming packets. Whenever the merge function receives a packet, it compares the desired field in the incoming packet (e.g., the fraction lost) against the maximum (or minimum) value seen so far. Once all upstream neighbors have been heard from, the resulting value (e.g., maximum loss rate) is forwarded toward the receiver (multicast sender).

Figure 6 shows (in pseudo-code) an example merge specification that computes the maximum values of the fields in the RR packet. For simplicity, we assume that each RR message contains only a single report. The *getTag* predicate simply returns a constant so that a node will merge all RRs that come in during a reporting interval. The *merge* function compares the incoming fields with the existing merge state s; all fields in s are initialized to 0. The flow variable G is the concast group identifier.

The *done* predicate returns true the first time it is called, and thereafter returns false. The *buildMsg* function sets a timer to automatically invoke itself (and the forwarding function) *interval* seconds from *now*. When the timer expires, it invokes forwardMsg(R,G,fsb.buildMsg(s)) to send the merged packet. This wakes up the merge spec *interval* seconds from *now* to forward the result of all RR packets seen during the next reporting interval. Thus, after the first packet, packets are forwarded every *interval* seconds.

Note that with this approach the multicast sender does not learn anything about (or need to manage/keep track of) the group's size or the identity of the group members. Because control information is merged as it travels toward the multicast sender, the multicast sender only receives a single RR packet, regardless of the group size. Moreover, control messages are only sent to the multicast sender and not to all group members, thus avoiding implosion at other group members. Clearly, other types of aggregation are also possible, such as computing the average jitter.

3.3. Multiplexing Receiver Reports

In certain cases, the application may want to discover the identity of all receivers, or associate a report with a particular receiver. To collect this information from large groups without causing implosion, the merge specification can be written to store RR packets for late aggregation. Fortunately, the RTCP packet format already supports the ability to concatenate multiple RR reports into a sequence of *report blocks* in an RR packet.

When a maximally sized RR packet has been constructed or a timeout interval has expired, the resulting packet is forwarded to the destination. Essentially, concast nodes perform packet processing similar to that of an RTCP translator [21]. The merge specification for concatenating RRs is shown in Figure 7 The *getTag* predicate (not shown) is the same as the one in Figure 6.

```
getTag($m$) {
    return(1);
}

merge($s, m, fsb$) {
    s.ssrc = G;
    s.frac_lost = max(s.frac_lost, m.src[0].frac_lost);
    s.cum_lost = max(s.cum_lost, m.src[0].cum_lost);
    s.highest = max(s.highest, m.src[0].highest);
    s.jitter = max(s.jitter, m.src[0].jitter);
    if (m.src[0].last_SR $>$ s.last_SR) {
        s.lastRR = m.src[0].lastRR;
        s.delay = m.src[0].delay;
    }
}

done($s$) {
  if (!s.started) {
      s.started = TRUE;
      return TRUE;
  } else
      return FALSE;
}

buildMsg($s$) {
  struct RR_packet pkt;

  pkt.src[0] = s;
  settimer(now+interval);
  return(pkt);
}
```

Figure 6. RR merge specification to compute maximums.

```
merge($s, m, fsb$) {
    for (i=0; i $<$ m.numrecords; i++) {
        s.src[fsb.count+i].ssrc = m.src[i].ssrc;
        s.src[fsb.count+i].frac_lost = m.src[i].frac_lost;
        s.src[fsb.count+i].cum_lost = m.src[i].cum_lost;
        s.src[fsb.count+i].highest = m.src[i].highest;
        s.src[fsb.count+i].jitter = m.src[i].jitter;
        s.src[fsb.count+i].lastRR = m.src[i].lastRR;
        s.src[fsb.count+i].delay = m.src[i].delay;
    }
    fsb.count += i;
}

done($s$) {
  if (fsb.count $>=$ 31)
      return(TRUE);
  else
      return(FALSE);
}

buildMsg($s$) {
  struct RR_packet pkt;
  canceltimer();
  fsb.count = 0;
  pkt = s;
  settimer(now+interval);
  return(pkt);
}
```

Figure 7. Merge specification to multiplex RRs (For simplicity the pseudo-code shown here ignores the case when the size of the incoming RR exceeds the capacity of the new RR).

3.4. Customized Receiver Reports

RTP also allows application-specific RTCP packet formats called APPs [21]. In this case, the application would create a customized merge specification that knows how to deal with the application-defined packet format. For example, the application may want to simultaneously compute maximum, minimum, and average loss rate experienced by receivers. In this case it might create an APP packet such as the one shown in Figure 8. The merge specification (not shown here) to compute/aggregate

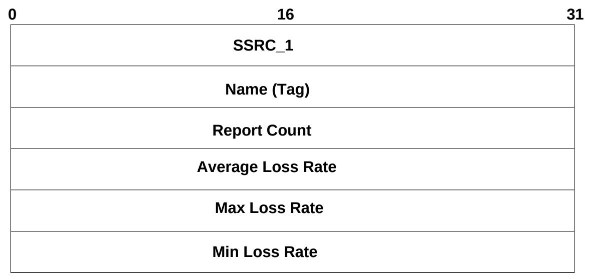


Figure 8. Example APP packet format.

these values would be similar to the code shown in Figure 6.

4. Scaling Multimedia Applications

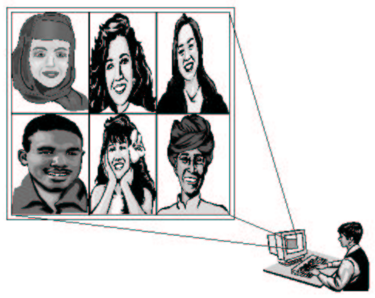
The previous examples primarily focused on ways to improve multimedia communication in the control plane. In this section we show how to implement scalable multimedia applications by using concast in the data plane. In particular, we show how to inject transcoding functionality into the network to dynamically manage bandwidth requirements and avoid network congestion. First, we describe our implementation of a video merging application that uses concast to intelligently transcode video streams to lower bit rates to prevent congestion and provide fair bandwidth sharing among video streams. Second, we describe a concast implementation of an audio application that dynamically combines music or voice exactly where needed (in the network), so that only a single audio stream crosses any bottleneck link.

4.1. Video Merging

Applications involving video often require the ability to receive video feeds from multiple sources simultaneously. Examples include applications such as distance learning and video monitoring/surveillance. The objective is to receive the best possible video quality from all sources. Because the video flows will compete for bandwidth at confluence points, not to mention competing with random cross traffic, it is difficult to devise end-to-end solutions that share bandwidth fairly among flows [16, 10].

Consider a distance learning application in which the instructor wants to see the video feed from each student in the class, while each student only needs to see the instructor. Video transmission from the instructor to the students can easily be handled by a multicast session originating from the instructor. However, the video flows from the students to the instructor quickly result in implosion and poor video quality if the bandwidth is not managed carefully.

(a) Initially there may only be four students who's video is merged into a single video stream that is displayed.



(b) As more people join the class, the concast merge function dynamically adjusts the video to make room for the new students.

Figure 9. Illustration of a Distance Learning Application: The video stream from each source is down-sampled at the merge point, resulting in the viewing window size at the receiver remaining constant while the number of sub-windows increases (i.e. more participants), and the size of sub-windows decreases.

To control the potential implosion at or near the instructor, a concast session can be established that transcodes the incoming streams into lower quality streams, thereby reducing the network bandwidth requirements. To support this type of application, we designed a simple merge function that scales the incoming video stream by down-sampling the pixels that comprise each frame of the video. The goal of the merge specification is to ensure that all incoming streams are down-sampled into a single outgoing stream. In other words, each network link should carry no more than one video steam. Consequently, the merge specification keeps track of the number of incoming streams and the number of students (video streams) encoded in each video stream. It then assigns a region of each outgoing frame to each incoming video stream and down-samples the stream appropriately to fit in the assigned region. As new students "join" the class, the other images are adjusted to make room for the new student (see Figure 9). Each composite stream carries information about how many original streams it contains, and how they have been last combined so that each node can determine how to combine its incoming streams.

To evaluate concast's ability to combine video flows inside the network, we implemented the video merge application using our (Linux-based) concast router implementation and measured its performance. Our test topology is shown in Figure 10. We used four video senders, each transmitting an unencoded (i.e., raw) video stream at a rate of 3 Mbps. All network links, except the link to the concast receiver, were 100 Mbps Ethernet. The link from the "Merge 1" node to the receiver was a 10 Mbps

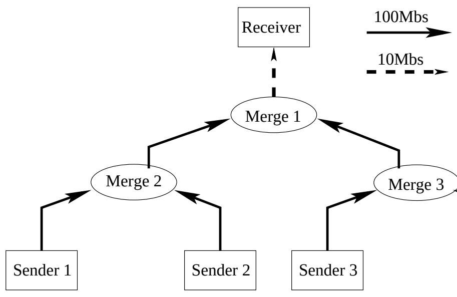


Figure 10. Topology used in the video merging experiments.

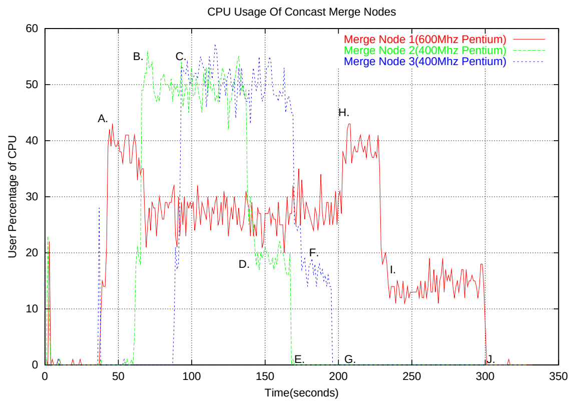
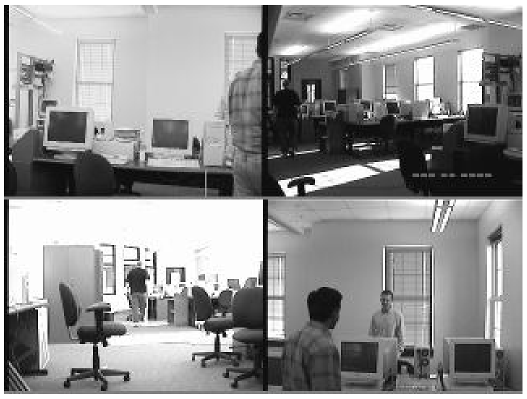


Figure 11. Merge processing load imposed on concast routers.

Ethernet to create a bottleneck. When all 4 senders join the group, the total (unicast) video bandwidth transmitted is roughly 12 Mbps, which leads to congestion over the 10 Mbps link to the receiver[‡]. For comparison purposes, we tested both a unicast and a concast implementation of the application.

Figure 11 shows the CPU load caused by merge processing on each of the three merging routers (nodes 1, 2, and 3) [§]. Our concast merge specification was written in Java and runs in a user-level JVM, which accounts for the majority of the load. Initially, video sources are started on senders 1 and

[‡]The realizable bandwidth is actually less than 10 Mbps.
[§]Merging node 1 happened to be a 600 Mhz Pentium, whereas the other nodes were 400 Mhz Pentiums – so the load appears to be lower on node 1 even though it is doing the same processing as the other nodes.


(a) Concast video application containing four merged streams.

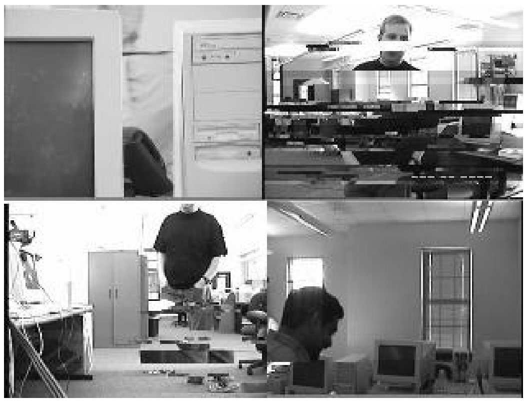

(b) Four unicast video streams with heavy loss(each originally 320x240 scaled to total 640x480 to fit figure).

Figure 12. Samples of frames taken from the experimental setup. The concast stream is 3Mbps, while unicast is four streams each at 3Mbps, with a bottleneck link of 10Mpbs. As a result the unicast streams are lossy. Lost packets are reflected in lost scanlines. In our application when a data is lost the corresponding scanline remains unchanged.

3. This causes the load on merging node 1 to increase (see point A in the graph). Video sources were then started on senders 2 and 4 (points B and C), causing the load to increase on merging nodes 2 and 3 respectively. This actually reduces the load on merging node 1 slightly because node 1 switches from vertical down-sampling to horizontal down-sampling. Due to the data structures used in the Java merging code, horizontal sampling involved simply ignoring a scan line of data, while in vertical sampling the algorithm iterated over a row of pixels. When senders 2 and 4 terminate the load returns (point H).

Note that when a node only has one upstream neighbor, packets are forwarded as normal without invoking the merge processing. Despite being implemented in java, the merging code (which is merging two 3 Mbps incoming streams into a single outgoing 3 Mbps stream) does not exceed a 60% CPU load. At the end of the test, the senders terminate (points D, F, and I) and after the concast softstate times out, the CPU loads again return to zero (points E, G, and J).

As a comparison, we also ran a test using only unicast transmission. In the unicast setup, all four senders sending video causes the implosion resulting in the complete loss of scanlines in one or more of the video feeds see Figure 12(b). In the concast setup, all four video windows receive data as smoothly as when only one sender is sending see Figure 12(a).

Figure 13 illustrates the bandwidth savings of merging video streams together. Packet traces were collected on the ingress interface of merging node 1 (the solid red line), and at the egress of merging

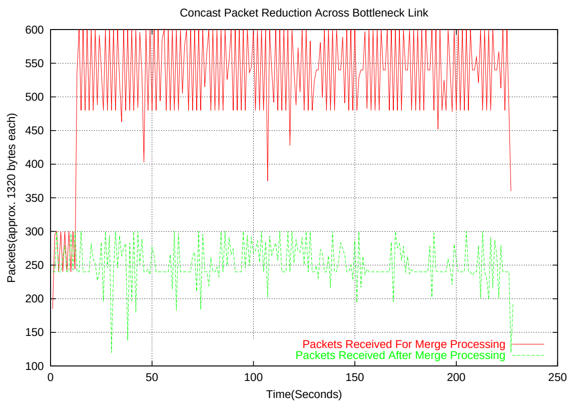


Figure 13. Bandwidth savings from the concast service

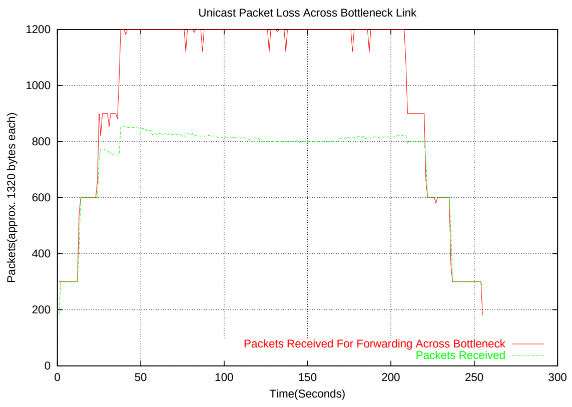


Figure 14. Unicast packet loss due to congestion

node 1 (the dashed green line). The graph clearly shows the merging of the two incoming streams cuts the traffic load in half as expected.

As a point of comparison, we made similar measurements using our unicast implementation. Figure 14 illustrates the incoming packet rate seen at Merge node 1 (the solid red line) and the outgoing packet rate (the dashed green line). Because the transmission exceeds the link's bandwidth, we see roughly 400 packet drops per second.

Finally, although this particular example did not require many-to-many video streams, other teleconferencing applications require the ability for all participants to see all other participants. This is a significant problem for conventional teleconferencing applications because, even when using multicast, it requires N video streams to every participant (one out and $N - 1$ in). However, by using concast to combine streams together, we can guarantee that end systems only need two video streams to participate (one going out, and one (combined) stream coming in). Streams going out would be merged enroute to a central aggregation point that re-multicasts the combined stream to all participants. Because data is merged inside the network, scalability of the central server is not a problem.

4.2. Audio Merging

A related application is audio mixing. Given N audio signals, standard audio mixing algorithms can be deployed at confluence points to reduce the bandwidth by a factor of N while still maintaining excellent sound quality. Such an algorithm might be desirable as part of a surveillance system. The following section describes an example audio merge specification and presents experimental results obtained from our prototype implementation.

We implemented an audio transcoding system of this type and measured its effectiveness using our concast framework. The idea is to convert N incoming audio streams, in our case 64 Kbps PCM audio streams sampled at 8000 Hz, into a single 64 Kbps outgoing stream that is a combination of the incoming voice channels. As a result, the destination machine receives a single 64 Kbps flow that contains the audio from all senders. A variety of audio mixing algorithms exist with some producing better sound-quality than others. We developed a basic audio mixing system (merge function) and implemented it on our concast framework. Our simple audio mixing function converts incoming μ-law streams to 14-bit linear samples, sums them, and converts the result back to a 64 Kbps μ-law stream; peaks that exceed the dynamic range are clipped. Although more sophisticated mixing functions are possible, the simple merge spec shown below is sufficient to evaluate the network performance.

Each outgoing packet contains a sequence number identifying the timing interval to which the packet corresponds. To ensure proper synchronization of the voice streams, each merge point maintains a variable n_i indicating the "next sequence number expected" from each incoming stream i. (Note that incoming streams may already have been merged upstream.) Every 125ms, the n_i's are incremented. When a packet arrives on stream i its sequence number p_i is checked against n_i. If the packet is late, the merged sample for its interval has already been transmitted; in that case $p_i < n_i$ and the packet is discarded. If $p_i > n_i$, the packet is early; it is buffered and merged into the proper interval. Because clocks drift, the combiner automatically adjusts sequence numbers for an upstream neighbor that consistently runs behind or ahead. If the incoming sequence number misses the expected value by 1 for some number of consecutive intervals, n_i is set to p_i.

We implemented the audio mixing algorithm as a merge specification which was then automatically deployed (via the concast service) across the test network shown in Figure 15.

In our experiments, senders collected 1000 audio samples into one packet every 125 ms. To test the service, we used a topology (Figure 15) with a bottleneck link and a legacy router to demonstrate backward compatibility with the existing Internet. In our topology G is a non-concast capable router. G_1 and G_2 are concast routers while G_3, G_4, G_5 and R are concast-capable end-systems. We created simultaneous audio flows by starting sender applications $S_1 – S_6$, each sending one stream toward R (a total of six 64 Kbps streams). The senders signal to join the concast flow, thereby triggering the Concast

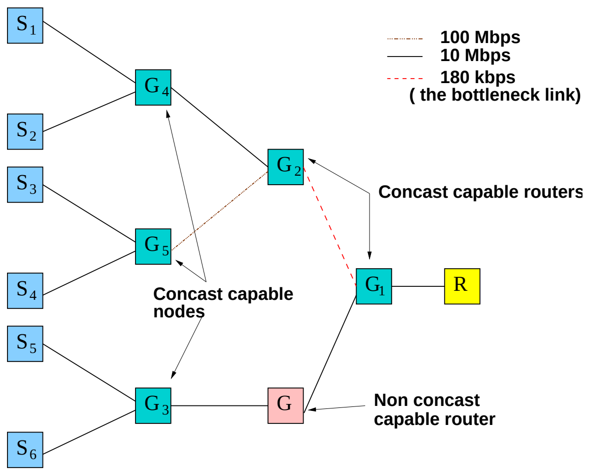


Figure 15. Experimental topology.

Signaling Protocol which downloads and installs the audio merge functions at the concast routers and nodes G_1 - G_5. The concast routers and nodes merge two streams sent by the upstream neighbors and forward one resultant merged audio stream towards R.

We compared the concast-based implementation with a unicast-only implementation, in which all streams are unicast to R. The merging of the audio streams was done at the receiver only.

Figures 16 and 17 show the senders whose data was successfully received at R over 125 ms intervals from 12.5 to 37.5 seconds for unicast and concast respectively. As seen in Figure 16, the unicast implementation imposed a load of 256 kbps on the bottleneck link, resulting in a high loss rate. These appear as gaps on the graph instead of a continuous line. However, we observe continuous line plots in the case of concast since merging at G_2, G_4 and G_5 ensured that the audio stream did not consume more than 64 kbps over the bottleneck link.

5. Related Work

Although many-to-one communication models arise frequently in practice, application writers have historically been forced to implement application-specific ad hoc solutions because the network offered no support for this type of communication model. Perhaps the most obvious examples of this are reliable multicast protocols which create overlay networks with special purpose nodes that merge or suppress feedback messages to avoid implosion [22, 23, 17, 14, 11, 11, 6].

As mentioned earlier, the RTP protocol tries to avoid implosion by reducing the receiver report transmission rate as the size of the group increases. Various heuristic techniques to estimate the group size with low overhead have been proposed [19, 20]. Although these statistical methods can reduce

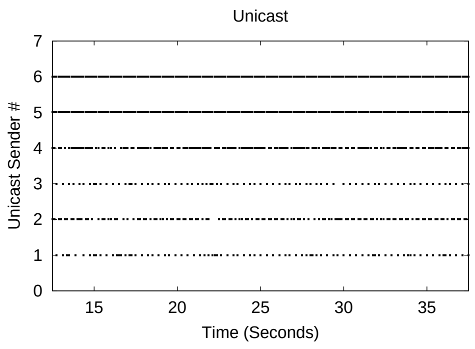


Figure 16. Unicast singals at R in each 125ms interval.

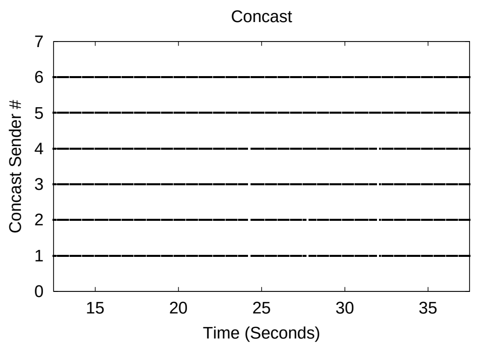


Figure 17. Concast singals at R in each 125ms interval.

the overhead needed to estimate group size, they do not address the problem of achieving scalability through anonymity; all receiver reports are still visible to the RTP source as opposed to providing the RTP source with only the information it needs (e.g., the maximum loss rate).

A recent trend in multimedia applications is to place transcoding servers at strategic points in the network [2, 3, 15]. The objective of these network servers is to maintain the highest possible quality while reducing the network load by either “thinning” the multimedia stream or by “combining” multiple streams together. Such approaches can provide additional scalability but are difficult to deploy because the locations where servers can be run is typically very limited, and identifying the “correct” location for a server is difficult and changes in response to changes in group membership.

Unlike application-level gateways, the approach we propose in this paper is based on a generic *network-level* service that is self-configuring, automatically instantiating the needed state and functionality only at nodes where it is required.

6. Conclusions

In this paper, we demonstrated how multimedia group applications can be implemented by end-systems in a scalable way using generic network level services like concast. We provided concrete examples and algorithms that show how a many-to-one service can be used to control feedback traffic in conjunction with multicast for multimedia delivery, and can also be used to perform the function of a transcoding gateway, combining audio and video streams to reduce bandwidth requirements.

We presented experimental results taken from our prototype implementation of audio and video applications implemented on (Linux-based) concast-capable routers. While programmable, ”active” services such as concast are not available in the Internet today, concast-capable routers could be deployed in private networks (e.g. a network of learning institutions), where services such as those described here are needed. More over, the service is backwards compatible with the existing Internet, allowing many of the advantages to be obtained even with partial deployment (e.g.of ingress/egress nodes of autonomous systems).

Because the concast framework allows users to define application-specific merge specifications, the single generic network-level abstraction is able to support widely different end-to-end uses. Our experimental results show that our video application, based on generic concast services, reduced packet loss by two orders of magnitude. Moreover, it ensured fair allocation of the bandwidth among senders. Although these types of generic services have been a long time coming, they offer significant benefits to a wide range of applications.

REFERENCES

1. The Linux Netfilter Project. http://www.netfilter.org.
2. The UCL Transcoding Gateway (UTG). http://www-mice.cs.ucl.ac.uk/multimedia/projects/utg/.
3. E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its application to Realtime Multimedia Transcoding. In *Proceedings of the ACM SIGCOMM '98 Conference*, Sept. 1998.
4. E. Amir, W. McCanne, and H. Zhang. An application level video gateway. In *ACM Multimedia '95*, 1995.
5. J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *ACM Sigcomm '94*, 1994.
6. B. Cain and D. Towsley. Generic Multicast Transport Services: Router Support for Multicast Applications. Technical Report CMPSCI TR 99-74, Umass, October 1999.
7. 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.
8. 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)*, 19(3):426–438, March 2001.
9. Tony Speakman et. al. Generic router assist (gra) for multicast transport protocols, July 2001. Internet Draft: draft-ietf-rmt-gra-arch-02.txt(work in progress).
10. R. Gopalakrishnan, J. Griffioen, G. Hjalmtysson, and C. Sreenan. Stability and Fairness Issues in Layered Multicast. In *Proceedings of the 1999 NOSSDAV Conference*, June 1999.
11. 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.
12. D. Katz. Ip router alert option, February 1997. Internet Request For Comments 2113.
13. I. Kouvelas, V. Hardman, and J. Crowcroft. Network Adaptive Continuous-Media Applications Through Self Organised Transcoding. In *Proceedings of the Network and Operating Systems Support for Digital Audio and Video Conference (NOSSDAV 98)*, July 1998.
14. L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *Proceedings of the INFOCOM Conference*, March 1998.
15. A. Mankin, L. Gharai, R. Riley, M.P. Maher, and J. Flidr. The Design of a Digital Amphitheater. In *Proceedings of the Network and Operating Systems Support for Digital Audio and Video Conference (NOSSDAV)*, Chapel Hill, NC, June 2000.
16. S. McCanne, V. Jacobson, and M. Vetterli. Receiver-Driven Layered Multicast. In *Proceedings of the ACM SIGCOMM '96 Conference*, October 1996.
17. 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).
18. S. Pingali, D. Towsley, and J. Kurose. A Comparison of Sender-initiated and Receiver-initiated Reliable Multicast Protocols. In *Proceedings of the ACM SIGMETRICS '94 Conference*, pages 221–230, 1994.
19. J. Rosenberg and H. Schulzrinne. Timer Reconsideration for Enhanced RTP Scalability. In *Proceedings of IEEE INFOCOM*, March 1998.
20. J. Rosenberg and H. Schulzrinne. Sampling of the Group Membership in RTP, February 2000. RFC 2762.
21. H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, January 1996. RFC 1889.
22. T. Speakman, D. Farinacci, S. Lin, and A. Tweedly. The PGM Reliable Transport Protocol, August 1998. RFC (draft-speakman-pgm-spec-02.txt).
23. R. Yavatkar, J. Griffioen, and M. Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *The Proceedings of the ACM Multimedia '95 Conference*, pages 333–344, November 1995.