Leveraging Emerging Network Services To Scale Multimedia Applications *

K. Calvert, J. Griffioen, S. Natarajan, B. Mullins, L. Poutievski, A. Sehgal, and S. Wen
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046

Abstract

Multicast services have been used for many years to transmit multimedia data to large receiver groups. However, 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 emerging services, specifically a *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 that show as much as two orders of magnitude in packet loss rates using these the generic services.

*Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0514. 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.

1 Introduction

Multicast provides data delivery to groups of receivers that can scale to very large sizes. Multicast allows applications to scale with group size by providing both *bandwidth savings*, by duplicating packets in the network instead of 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 long been prime 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 to send information in the reverse direction in a multicast application (i.e. from many group members to one or a few end systems) is via the same service used in the forward direction, i.e. unicast or multicast. For example, many deployed multimedia applications need control information (feedback) to be sent from 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* [17, 18]; 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 [18, 19, 4], or they create an overlay network in the reverse direction to aggregate control information [22, 16].

Recently, however, various network-based mechanisms have been proposed to aid in overcoming these limits. For example, Pragmatic General Multicast [21] is a protocol based on a set of router-based mechanisms that can improve the scalability of multicast applications that use sender-based retransmissions. Generic Router Assist [7] comprises simple router-based mechanisms that can be used in a variety of ways to improve scalability and performance of group applications. Concast [8, 9] 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.

In this paper, we show how such network-based mechanisms 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 [3, 2, 12]. 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 brief overview of the concast service on which we have developed solutions to the problems described earlier. Section3 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 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 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.[1] 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 maintain using soft-state techniques similar to RSVP [5].

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.

The merge semantics are characterized by a *merge specification*, which defines (1) how data-

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

grams 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 1(b) which will be executed in the framework shown in Figure 1(a).

In all the examples described in this paper, the language used to supply and execute merge specifications was the Java language. Datagrams are processed hop-by-hop at concast-capable nodes via a new IP option called the *concast ID* which hold the group identifier G. Given the pair (R, G) the packet is demultiplexed to the appropriate merge function.

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. All results shown in this paper were collected using this real implementation of the service.

3 Scaling RTP/RTCP

A common protocol used in many multimedia video and audio applications is the *Real Time Protocol* (RTP) [20]. RTP is a general purpose protocol that can be used by a wide range of realtime applications. Because realtime applications often require feedback from the participants in the multicast group, RTP's control protocol, RTCP, provides a generic feedback mechanism via the use of 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 2. Each RR corresponds to a single reporting interval and carries

```
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) ) {
           (s, m) = fsb.buildDatagram (s);
           FORWARD_DG (fsb, s, m);
       }
       PUT_MERGE_STATE (fsb, s, t);
   }
}
```

(a) Network 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 getTag (m) = getTag (m')

merge (s,m,f) : the function that combines messages togather. 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 state s, and returns the payload to be forwarded toward the receiver.

(b) Functions that comprise a Merge Spec

Figure 1: The Concast Framework

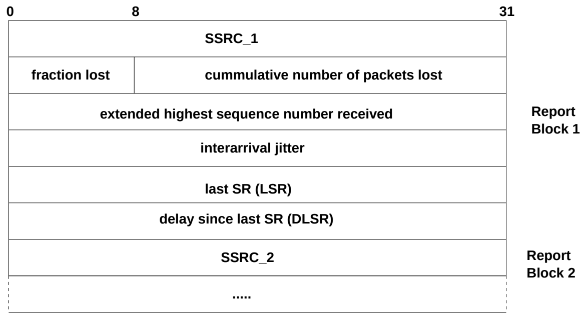


Figure 2: Format of Report Blocks in the Receiver Report

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), interpacket delays (jitter), and information identifying the reporting interval. Each RR may contain reports from multiple sources.

Each receiver in the group periodically transmits a RR describing its current performance information. As the group size increases, the number of RRs transmitted grows and implosion becomes a real 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 required 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 [18, 19].

Using concast for RR transmission has several obvious advantages, the main one being that feedback messages only need to be sent (via concast) to the realtime data source. Other nodes can remain oblivious to the size and membership of the group and can send 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 3.

3.1.1 Aggregating Receiver Reports

Generally a multicast sender is not interested in each receiver's individual report, but rather wants to know quantities 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

```
struct RR_Packet {
  struct report src[32]; /* may contain multiple reports   */
}

struct report {
  u_int32 ssrc;          /* receiver generating this report */
  u_int32 frac_lost:8;   /* fraction lost                   */
  u_int32 cum_lost:24;   /* total (cummulative) lost        */
  u_int32 highest;       /* highest seq # received          */
  u_int32 jitter;        /* interarrival jitter             */
  u_int32 lastRR;        /* last receiver report            */
  u_int32 delay;         /* delay since last RR             */
}

struct RR_Packet *m;      /* message m points to an RR_Packet */
```

Figure 3: RR Packet Fields

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 4 shows (in psuedo-code) an example merge specification that computes the maximum values of the fields in the RR packet.

For simplicitly, the following code requires that each RR message contains 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

```
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 4: RR merge specification to compute maximums.

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.

3.1.2 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 hold RR packets for aggregation as they arrive. 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 on to the destination. Essentially, concast nodes perform packet processing similar to that of an RTCP translator [20]. The merge specification for concatenating RRs is shown in Figure 5[2]. The *getTag* predicate (not shown) is the same as the one in Figure 4.

```
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 5: Merge specification to multiplex RRs.

3.1.3 Customized Receiver Reports

RTP allows application-specific RTCP packet formats called APPs [20]. To use concast 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 6. The merge specification (not shown here) to com-

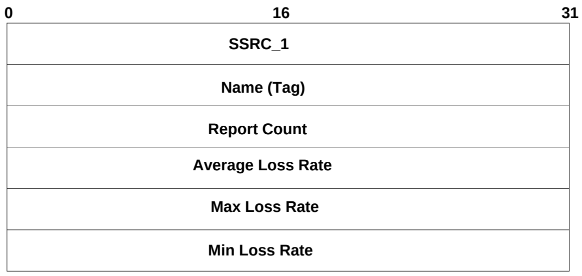


Figure 6: Example APP packet format.

[2] For simplicity the psuedocode shown here ignores the case when the size of the incoming RR exceeds the capacity of the new RR.

pute/aggregate these values would be similar to the code shown in Figure 4.

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 will 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 implemenation 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 [15, 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.

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, other students images are adjusted to make room for the new student (see Figure 7).

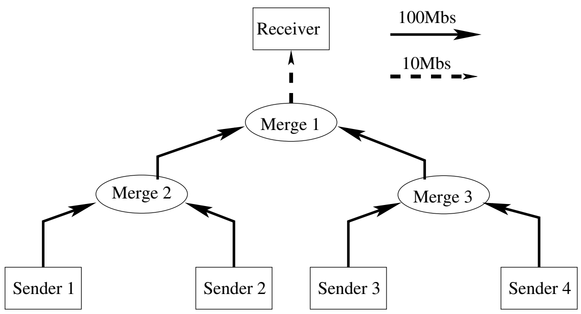
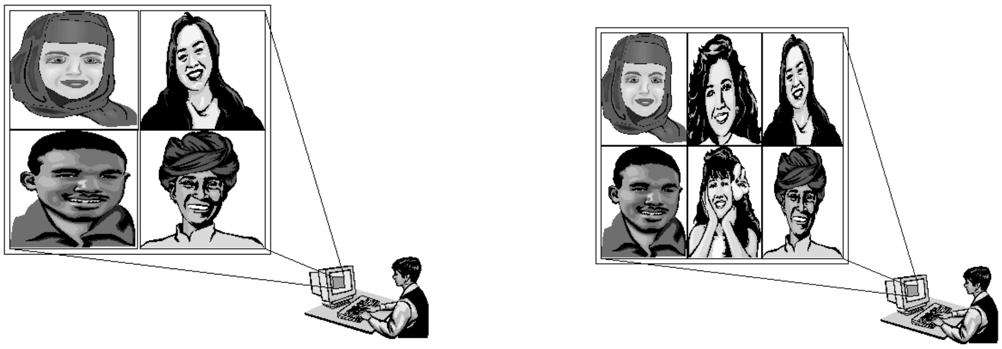


Figure 8: Topology used in the video merging experiments.

To evaluate concast's ability to combine video flows inside the network, we implemented the video merge application on top of our (Linux-based) concast router implementation and measured its performance. Our test topology in shown in Figure 8. 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



(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 joins the class, the concast merge function dynamically adjusts the video to make room for the new students.

Figure 7: 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 subwindows increases (i.e. more participants), and the size of subwindows decreases.

the "Merge 1" node to the receiver was a 10 Mbps 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[3]. For comparison purposes, we tested both a unicast and a concast implementation of the application.

Figure 9 shows the CPU load caused by merge processing on each of the three merging routers (nodes 1, 2, and 3)[4] 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 3. This causes the 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

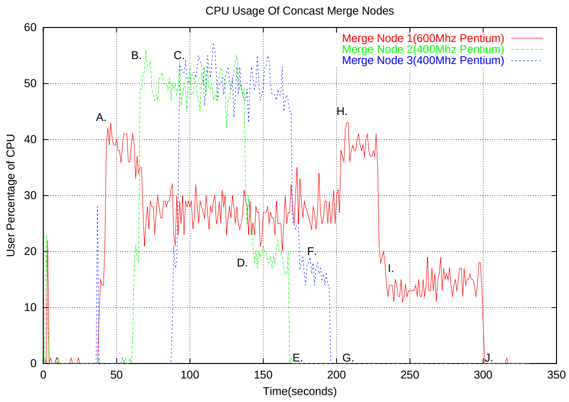


Figure 9: Merge processing load imposed on concast routers.

[3]The realizable bandwidth is actually less than 10 Mbps.

[4]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.

2 and 3 respectively[5]. Note that when a node only has one upstream neighbor, packets are forward 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 with only unicast transmission under the same setting. In the unicast setup, when all four senders start sending video, the receiving windows randomly experience discontinuation in the video stream due to packet losses. In the concast setup, all four video windows receive data as smoothly as when only one sender is sending.

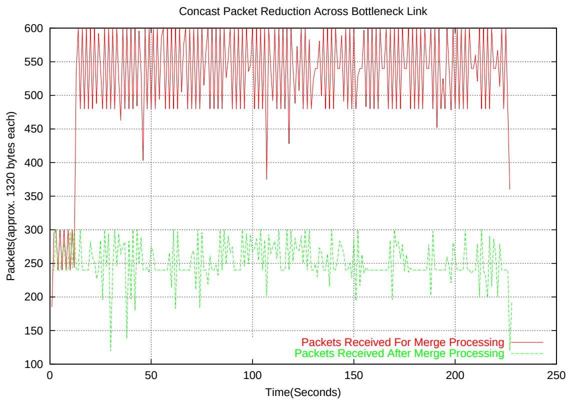


Figure 10: Bandwidth savings from the concast service

Figure 10 illustrates the bandwidth savings of merging video streams togeather. Packet traces were collected on ingress to merging node 1 (top), and at the egress of merging node 1 (bottom). The graph clearly shows the merging of the two incoming streams into a single outgoing stream.

As a point of comparison, we made similar measurements on our unicast implementation. Figure 11 illustrates the incoming packet rate to 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.

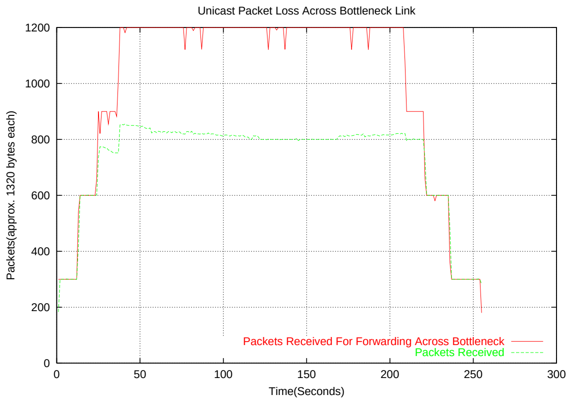


Figure 11: Unicast packet loss due to congestion

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 remulticasts 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. The following section describes an example audio merge specification and presents

[5]This actually reduces the load on merging node 1 slightly because node 1 switches from horizontal downsampling to vertical downsampling (which is a better match for the data structures used.) When senders 2 and 4 terminate the load returns (point H).

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 the 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 show in Figure 12. The merge specification for the audio mixing application is shown below.

```
getTag(m) {
    /* all packets have the same tag in this time slot
    merge will check the sender id and sequence number */
    return(1);
}

merge(s, m, fsb) {
    /* haven't seen a packet from this sender yet */
    if !isRecvd(m.src, fsb)
        /* converting incoming to 14-bit linear samples and
        summing them into the current merge state. */
        s = convert(s, m);
    else
        drop(m);
}

done(s) {
    if !timerSet()
        /* schedule packet to be built and sent 125 ms from now */
        schedule(build_n_forward(s), 125);
    return(FALSE);
}

buildMsg(s) {
    /* convert the current merge state back to a 64 Kbps
    stream, clipping peaks that exceed the dynamic range. */
    pkt = buildPkt(s);
    return(pkt);
}
```

In our experiments, senders collected 1000 audio samples into one packet every 125 ms. To test the service, we use a topology (Figure 12) 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 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

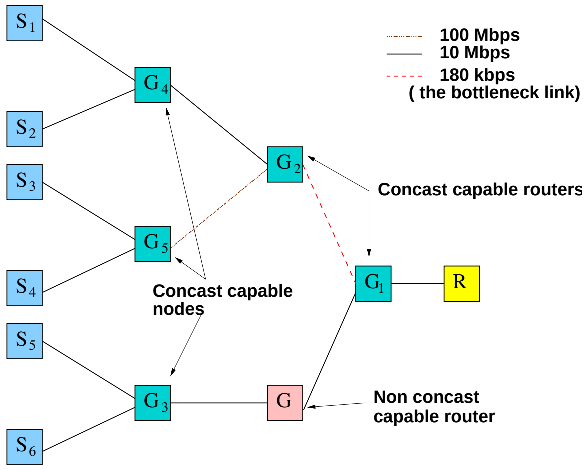


Figure 12: Experimental topology.

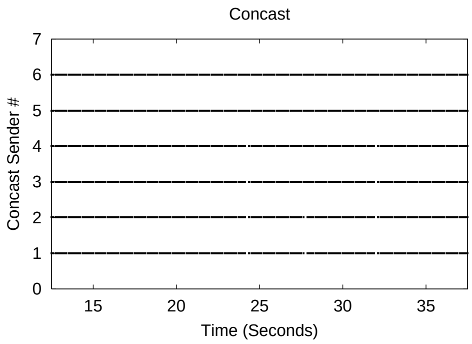


Figure 14: Concast Senders' signals played out at R in each 125 ms interval.

streams are unicast to R. The merging of the audio streams was done at the receiver only.

Figures 13 and 14 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 13, 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.

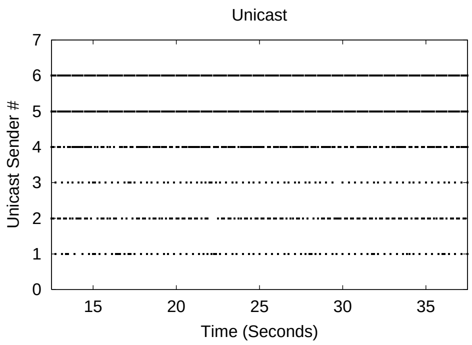


Figure 13: Unicast Senders' signals played out at R in each 125 ms interval.

5 Related Work

Although many-to-one communication models arise frequently in practice, application writers have historically been forced to implement application-specific adhoc solutions because the network offered no support 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 [21, 22, 16, 13, 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 hueristic techniques to estimate the group size with low overhead have been proposed [18, 19]. Although these statistical methods can reduce the overhead needed to estimate group size, they do not address the problem of achieving scalability through anonymity; all receiver reports 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 [1, 2, 14]. 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 result taken from our prototype implementation of audio and video applications implemented on (Linux-based) concast capable routers. Because the concast framework allows users to define application-specific merge specifications, the single generic network-level abstraction was able to support widely different end-to-end uses. Our experimental results show that our video application, based on a 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 service have been a long time in coming, they offer significant benefits to a wide range of applications.

References

[1] The UCL Transcoding Gateway (UTG). http://www-mice.cs.ucl.ac.uk/multimedia/projects/utg/.

[2] 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.

[3] E. Amir, W. McCanne, and H. Zhang. An application level video gateway. In *ACM Multimedia '95*, 1995.

[4] J.-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *ACM Sigcomm '94*, 1994.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReserVation Protocol – Version 1 Functional Specification, September 1997. RFC 2205.

[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] Brad Cain, Tony Speakman, and Don Towsley. Generic router assist (gra) for multicast transport protocols, March 2000. Internet Draft: draft-ietf-rmt-gra-arch-01.txt.

[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] 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.

[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] 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.

[13] L. Lehman, S. Garland, and D. Tennenhouse. Active Reliable Multicast. In *Proceedings of the INFOCOM Conference*, March 1998.

[14] 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.

[15] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-Driven Layered Multicast. In *Proceedings of the ACM SIGCOMM '96 Conference*, October 1996.

[16] 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).

[17] 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.

[18] J. Rosenberg and H. Schulzrinne. Timer Reconsideration for Enhanced RTP Scalability. In *Proceedings of IEEE INFOCOM*, March 1998.

[19] J. Rosenberg and H. Schulzrinne. Sampling of the Group Membership in RTP, February 2000. RFC 2762.

[20] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, January 1996. RFC 1889.

[21] T. Speakman, D. Farinacci, S. Lin, and A. Tweedly. The PGM Reliable Transport Protocol, August 1998. RFC (draft-speakman-pgm-spec-02.txt).

[22] 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.