# Concast: Design and Implementation of an Active Network Service

Kenneth L. Calvert, James Griffioen, Billy Mullins, Amit Sehgal, and Su Wen

*Abstract*— **Concast is a network layer service that provides many-to-one channels: multiple sources send messages toward one destination, and the network delivers a single "merged" copy to that destination. As we have defined it, the service is generic but the relationship between the sent and received messages can be customized for particular applications. In this paper we describe the concast service and show how it can be implemented in a backward-compatible manner in the Internet. We describe its use to solve a problem that has eluded scalable end-system-only solutions: collecting feedback in multicast applications. Our preliminary analysis of concast's effectiveness shows that it provides significant benefits, even with partial deployment. We argue that concast has the characteristics needed for a programmable service to be widely accepted and deployed in the Internet.**

*Keywords*— **concast, active networks, scalable multicast**

## I. Introduction

The Internet offers two types of best-effort channels to applications: point-to-point *unicast* channels and point-to-multipoint *multicast* channels. Although multicast service is not universally available in the Internet today, the value of a channel that enables one sender to reach a large group of receivers via a single sending operation has long been recognized. Point-to-multipoint channels make scalable applications possible through abstraction; a group of receivers of arbitrary size is represented by a single identifier—a multicast address. A large number of multicast-based applications have been envisioned, and a good many have been deployed.

Symmetry considerations suggest that the Internet should support a third type of channel (multipoint-to-point), but it currently does not. Such a channel would provide a form of abstraction analogous to multicast, enabling a single receiver to treat a large group of senders as a single entity. The absence of such a channel gives rise to the problem of *implosion* in large-scale multicast applications. When information needs to flow in the direction opposite the multicast channel, the only option is to use multiple unicast channels. The multicast source is then forced to deal with receivers individually instead of as a group, and network resources near the source can become overloaded due to the "funneling" of traffic. In this paper, we propose and describe *concast*, a backward-compatible Internet service that provides a general form of scalable many-to-one channel.

What should be the semantics of a multipoint-to-point channel? In a multicast channel the network performs duplication, delivering identical copies of a datagram to different receivers. To provide a symmetric inverse channel, the network should *un-duplicate* datagrams. That is, when multiple senders (members of the concast group) transmit identical datagrams to the single receiver, at most one copy is delivered. We call this the *simple concast* service. The duplicate suppression capability of simple concast is useful whenever a receiver only needs an indication that *some* sender sent a message, as is the case for example with retransmission requests. In the general case, however, a receiver communicating with many senders may need some other function of the sent messages—for example, an indication that *all* senders transmitted a given message, or an upper bound on the values contained in the messages.

To accomodate diverse application needs without building application-specific functionality into the network, our concast service is *customizable* to the needs of an application. In particular, we allow the *receiver* to specify the mapping from sent datagrams to delivered datagrams, via code that conforms to a predefined programming interface. Other approaches are possible—for example, the network might offer a set of predefined mappings from which the receiver must select at channel setup time—but defining the service in terms of a programming interface is the most general, and also presents the most interesting challenges in terms of scalability and protection of the integrity of the network.

Concast is similar to multicast in some ways: users signal the network before using the service, and concast datagrams must be recognized as such by routers in order to be processed en route to their destination. On the other hand, concast differs from multicast in that it is incrementally deployable, and does not require any routing or addressing capabilities beyond unicast forwarding. As we shall show, concast can provide benefits in terms of scalability even when supported by only a few routers in the network.

The rest of this paper is organized as follows. Section II considers potential applications of concast. Sec-

Effort sponsored in part by the National Science Foundation and EPSCoR Grant agreement EPS-9874764 and by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory under agreement number F30602-99-1-0514. 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, AFRL, or the U.S. Government.

tion III describes the concast *service* and the steps taken by senders and receivers to use it. Section IV describes the components needed to implement the service, including the hop-by-hop processing of concast packets and the concast signaling protocol (CSP). This section also presents an overview of our prototype implementation. Section V discusses the use of concast in collecting feedback for multicast applications. Section VI addresses the significant question of how such a programmable service may affect the network. Section VII considers the benefits of partial deployment of concast. In conclusion, in Section VIII we summarize the characteristics of concast in view of some necessary conditions for deployment of new services in the Internet.

## II. Applications of Concast

The programmability of concast makes it useful in a wide range of contexts. In this section we first describe an especially prominent problem for which concast is a natural solution: scalable feedback for multicast applications. Some have proposed incorporating various functionality into the network to support such applications. In contrast, concast is a generic service that provides needed functionality for multicast applications, but can also be used by applications that do not involve multicast. Later in the section we list some of these applications.

### A. Feedback in Multicast Applications

Distributed applications involving large numbers of participants are increasingly common and important in the Internet. Scaling such applications to support large groups requires abstractions such as those provided by the Internet multicast service (IP multicast). IP multicast represents a group of receivers of arbitrary size with a single address, and frees the sending application from the task of keeping track of group membership. This form of information hiding is key to scalability.

Large-group applications often require that some form of feedback flow from receivers to sender. In many cases the feedback recipient is *not* interested in individuals' detailed information, but as we have observed in the Introduction, it is forced to deal with it because feedback must be sent via unicast.

Consider the feedback required in applications such as shared whiteboards [1], [2] and other "app-sharing" tools [3], [4], [5] that that need to reliably multicast data. Reliable multicast delivery is typically implemented using positive or negative acknowledgments. In the case of positive acknowledgements, the source is generally only interested in whether it has received an acknowledgement from *every* receiver. Ideally the network would forward an acknowledgement along a path toward the source only when one has been received along all incoming paths from receivers. Similarly, for negative acknowledgements, the question is whether *some* receiver sent one. Thus at most one negative acknowledgement need be forwarded along any path from receiver to source; others may safely be suppressed by the network.

Another common form of feedback is performance information such as loss rate, observed jitter, or highest sequence number received. This kind of feedback information is often used by a multicast source to adapt its transmission to present network conditions and group membership, as in RTP [6] and AER/NCA [7], [8]. In such applications the feedback recipient is not really interested in individuals' data, but in such summary quantities as the maximum loss rate, the minimum delay, and the minimum and maximum sequence numbers received. Collecting this type of information is a well-known problem for protocols like RTP/RTCP [9], [10]. In RTCP, the rate at which feedback packets are sent depends on the number of group members. As the group size increases, the rate decreases. This has the unfortunate side-effect of reducing the reaction rate (time) of the sender. Moreover, to learn the group size, RTCP requires that "Receiver Report" feedback messages be multicast to the entire group, not just the sender. Again, a feedback service that can be programmed to select the maximum loss rate or the minimum video quality can provide the application-specific functionality required to scale to large numbers of receivers.

In summary, a scalable feedback service benefits application performance, but perhaps more importantly simplifies the coding process, freeing the application writer from dealing with group membership concerns. Indeed, such a service would make many large-scale group applications viable. However, because different applications need different kinds of feedback, the service needs to be customizable, adapting to the semantic requirements of the application (e.g., MIN, MAX, SUM, duplicate supression, acknowledgement aggregation, etc) in the way messages are "merged" together. Later in the paper we show how to build a scalable feedback collection service using our concast framework.

### B. Other Concast Applications

The following broad classes of applications can be expected to benefit from concast services:

- **Group-request-Group-reply Applications:** In these applications, a single machine issues a request to a group of machines (typically via multicast) and then waits for a response from all group members. This model of interaction occurs both at the network level and the application level. Many distributed applications require end-to-end (application-level) acknowledgement messages to ensure reliability [11].
- **Report-in** Applications: These applications arise in a variety of settings involving distributed sensors, robots, or other devices that periodically transmit data to a centralized monitoring and control system. Examples include networks of surveillance cameras used for security/tracking, sensors in manufacturing quality control systems, and stereo reconstruction from multiple viewpoints. In some cases, the signals only become meaningful when combined and processed.

• **Client-Multiserver and Multiclient-Server Applications:** To ensure reliability or availability, *client-multiserver* applications replicate server functionality across multiple machines. Client requests are multicasted to the servers who send simultaneous responses to the client. The client often needs only one, or $K$ out of $N$, responses. For example, a distributed database may require $K$ out of $N$ acknowledgements when storing replicas of a newly written record. We use the term *multiclient-server* to represent the classical client-server model, in which a single server responds to request from any number of client machines. Concast offers benefits whenever concurrent requests are similar or even the same (e.g., URL request to a web server).

• **Collaborative/Multimedia Applications:** Real-time conferencing and application sharing can benefit from concast communication when they require synchronization (e.g., floor control) and/or resolution of inputs into a single stream or program. Several systems already provide audio mixers that merge the voice streams of conference call participants.

## III. Internet Concast Service

In this section we describe the concast *service* and the steps taken by applications to use it. For the purposes of this discussion we treat the network as a monolithic cloud, ignoring the existence of multiple nodes.

A *concast flow* is defined by a pair $(R, G)$, where $R$ is the receiver's (unicast) IP address and $G$ is the *concast group identifier*. The senders and receiver must agree on $G$ in advance, using some out-of-band means. Like multicast addresses, concast group identifiers are chosen from a flat address space and are carried in the Internet Protocol header. If concast were completely symmetric with multicast, the concast group identifier would be placed in the source address field of the IP header, and a concast datagram would appear to come from the concast group instead of any particular node. However, in our design the group identifier is carried in a *Concast Group Identifier* option, while the IP source address field contains a legitimate (unicast) address.[1] One reason for using a real address in the source field is to avoid problems with anti-spoofing reverse path checks, which have been deployed in many parts of the Internet. Another is that ICMP messages can be sent in response to concast packets that encounter problems. Finally, a non-address group identifier does not consume any unicast or multicast IP address space, and can be of any size and form that is convenient.

The normal sequence of events for establishing and using a concast flow is as follows. For the time being, the simple concast (duplication suppression) semantics described in the Section I are assumed.

1. The receiving application program (at host $R$) initiates a flow by signaling its local concast implementation giving the desired group identifier $G$ and a *merge specification.* This signaling might be done by setting a socket option. (Details of the merge specification are discussed in the next subsection.) At this point, the network is unaware of the flow and it consumes no network resources.
2. To join the flow, a sending application program requests its local concast module to join the flow identified by $(R, G)$. This causes the concast module at sending host $S$ to signal the network, which in turn checks for existence of a flow $(R, G)$. Finding none, the network signals the concast module at $R$ to confirm the existence of a flow with the given ID and get its merge specification.
3. $R$'s concast module replies with the merge specification, which is then installed in the network; the network informs the concast module at $S$ (which in turn informs the sending application program) that it is now a member of the specified concast flow. When other programs signal the network to join flow $(R, G)$, the network already has the merge specification so their requests are answered immediately.
4. Sending programs transmit IP datagrams as usual. The local IP/concast implementation ensures that each datagram sent on the concast flow carries a concast option containing $G$.
5. When a concast datagram is sent, the network recognizes it as such and processes it according to the merge specification and any stored state information for $(R, G)$. For simple concast, the state indicates which payloads have already been delivered to $R$. If the current payload is identical to one that has already been delivered, the current datagram is discarded. Otherwise, it is recorded and delivered to $R$.
6. The receiving application receives packets via the network API in the traditional way.

Senders may leave the group at any time by signaling their local concast module. When the receiving application program signals its local concast implementation to end the flow (for example, by closing the receiving socket), the concast module signals the network to indicate that the flow no longer exists. The network, in turn, informs all senders who have joined the flow that it has gone away.

The simple concast service is not feasible as described, because it requires that the network remember the payload of every IP datagram ever sent on a flow. However, a weaker form of simple concast can be implemented by specifying a bound $B$ on the number of packets the network "remembers" for a flow at any time, and storing only a small hash or fingerprint of the payload. The weaker form of the service is indistinguishable from "ideal" simple concast so long as no more than $B$ distinct payloads belonging to a flow are in the network simultaneously and hash collisions are rare; yet the weaker form requires only a bounded amount of per-flow storage. Later, in Section VI, we discuss other limits on the resources consumed by merge computations.

In reality the network is not a single monolithic entity; concast must be implemented hop-by-hop at individual routers as packets are forwarded from source to destination. This hop-by-hop processing is described in the next

[1] An earlier design did carry the concast address in the source field [12].

section, where we present details of the concast implementation including the programming API.

## IV. Implementing Concast

Implementing concast requires the following components:
- Modifications to the IP forwarding process of concast-capable routers to recognize concast datagrams and apply merge processing hop-by-hop.
- A merge processing framework defining the processing applied to each concast datagram at each concast-capable router. This processing consists of a fixed part, which is the same for all flows, and a customizable part, which is defined by the merge specification supplied by the concast receiver.
- The concast signaling protocol, which governs the distribution of flow-related state information (including the merge specification) to the relevant parts of the network.

In the following sections we discuss these components in turn. Then we briefly describe our prototype implementation.

### *A. Identifying Concast Packets*

Concast-capable routers have to recognize concast datagrams and divert them from the normal forwarding processing for merge processing. Unlike multicast datagrams, concast datagrams cannot be distinguished from regular datagrams by their destination address; some other mechanism is required to trigger concast processing. Fortunately this is something concast has in common with other protocols, such as RSVP, which also need packets to be processed at intermediate nodes on the way to a destination. The *Router Alert* IP option [13] is defined for the specific purpose of flagging datagrams for closer examination en route. The presence of Router Alert in the IP header causes a router, during forwarding, to apply checks for special cases such as a particular payload type (e.g., RSVP or the Concast Signaling Protocol) or, for concast data packets, the presence of a second option containing the concast group ID.

The actual path between two concast-capable nodes may include multiple concast-oblivious nodes, which treat concast packets as regular IP datagrams. The presence of Router Alert may cause datagrams to be taken off the "fast path" at concast-oblivious nodes, but it does not interfere with forwarding.

### *B. Hop-by-Hop Processing*

Concast-capable nodes maintain state for each active concast flow. The *Flow State Block* (FSB) contains the merge specification, state information for any in-process merges, and an *Upstream Neighbor List* (UNL) containing the addresses of the node's "children" in the concast tree. Each entry in router $n$'s UNL identifies a concast-capable node (either an end system hosting a sending application, or a router) for which $n$ is the next concast-capable hop on the unicast path toward $R$. The Upstream Neighbor List is maintained by the Concast Signaling Protocol, described in the next subsection. It is used to propagate information upstream, toward the senders in the concast group, for example when the receiver terminates the flow.

When a concast datagram is recognized and passed to the merging framework for processing, the flow state block for the $(R, G)$ flow, if any, is retrieved. If there is no FSB for the flow, the packet is silently dropped. Otherwise, the packet is processed according to the flow's merge specification as described next.

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. For example, in the inverse multicast service described above, two packets would be in the same DEC if they have the same hash or fingerprint. Because packets are processed one at a time as they arrive, each concast node maintains a *merge state block* for each active datagram equivalence class of a flow. To limit the amount of per-flow state, the size and number of the active merge state blocks must be limited by the merge processing framework as discussed earlier for the case of simple concast.

When merge processing is completed for a particular DEC, a concast datagram is constructed using information in the merge state block, and forwarded to $R$. The IP header of this forwarded datagram has $R$ in the destination address field and the IP address of (the outgoing interface of) the node doing the processing in the source address field. The protocol number, fragmentation information, transport header, and IP payload of the forwarded datagram are defined by the merge specification. The Time-to-Live (TTL) field of the IP header contains a value computed from the TTL values of incoming datagrams belonging to the same datagram equivalence class, in a manner discussed below.

From the foregoing discussion, it should be clear that the merge specification must include the following information:
- A description of the merge state block.
- A mapping from concast datagrams to datagram equivalence classes.
- A description of the actual "merge" computation, in a form that maps an arriving message and a merge state block into an updated merge state block.
- A description of the conditions under which merging is considered complete.
- A description of the message to be forwarded upon completion, in the form of a mapping from merge state to concast datagram payload.

The most general approach (and the one we have implemented) is for the receiver to supply code for the following data types and functions:

`struct MergeStateBlock`: A declaration of the state block (data structure) maintained for each active DEC.

`DECtag`*getTag*(`ConcastPkt` $m$): A *tag extraction* function returning a "key"; the key is unique to the DEC to which a packet belongs. Messages $m$ and $m'$ are in the same DEC iff $getTag(m) = getTag(m')$. The fixed part of

```
void ProcessMessage(Receiver R, Group G, ConcastPkt m) {
  FlowStateBlock fsb;
  DECTag t;
  MergeStateBlock *s;

   fsb := LOOK_UP_FLOW(R,G);
   if (fsb != NULL) {
     t := fsb.getTag(m);
     s := GET_MERGE_STATE(fsb,t);
     s := fsb.merge(s,m,fsb);
     if (fsb.done(s)) {
       (m,s) := fsb.buildMsg(s);
       FORWARD_MSG(R,G,m);
     }
     PUT_MERGE_STATE(fsb,t,s);
   }
} /* ProcessMessage */
```

Fig. 1. Per-packet merge processing.

the merge processing framework allows instances of type `MergeStateBlock` to be stored and retrieved using these keys.

`MergeStateBlock` *merge*(`MergeStateBlock` $s$, `ConcastPkt` $m$, `FlowStateBlock` $f$): The merge computation itself. The first parameter refers to the `MergeStateBlock` 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 structure is the same for all flows and is accessed read-only. (It contains information, such as the Upstream Neighbor List, which may be needed by the merge computation.) The function returns the updated `MergeStateBlock`.

`boolean` *done*(`MergeStateBlock` $s$): The *forwarding predicate* that decides whether a message should be constructed (by calling *buildMsg*) and forwarded to the receiver.

(`ConcastPkt`,`MergeStateBlock`) *buildMsg*(`MergeStateBlock` $s$): The *message construction* function, which takes the current merge state and returns a pair containing the concast IP datagram to be forwarded toward the receiver and the updated merge state. The concast datagram contains IP and transport headers and payload; the source, destination and TTL fields of the IP header are overwritten by the merge framework, and the Router Alert and Concast Group ID options are added before the datagram is forwarded. The destination address of the resulting IP datagram is $R$, the source address is the address of the outgoing interface toward $R$, and the TTL value is guaranteed to be one less than the minimum incoming TTL value of any message in this DEC.

Pseudocode for the generic *fixed* part of the hop-by-hop processing, which invokes the above functions, is shown in Figure 1. The functions in capital letters are built-in parts of the merging framework having the obvious semantics. The merging framework must isolate different flows' merge computations from each other, for example by running each in its own "sandbox".

The utility of the concast service is to some extent determined by the language in which the merge specification is defined, and the built-in facilities that can be invoked by the above functions. For safety reasons the functions making up the merge specification need to be restricted to local computations that do not involve communication with the outside world. They should therefore be supplied in an encoding that allows their restricted functionality to be guaranteed. One possibility is to allow the functions to be supplied in a mobile code language, e.g. as Java bytecodes, and to execute them in an environment that allows only access to "safe" facilities. Facilities considered safe might include various local variables that can be accessed on a read-only basis, for example an accurate time source. Read-only access to information about the flow, such as the Upstream Neighbor List, should also be provided.

Our implementation (Section IV-D) also allows the *merge*() function to arrange for the last part of the processing loop (i.e. the *done*() conditional block) to be invoked with a particular merge state block after a *timeout* period. This capability makes it possible to ensure that a message is forwarded at some point in spite of losses (otherwise messages are forwarded only in response to a message arrival). A flow is allowed to have at most one pending timeout associated with each active merge state block (i.e. each DEC).

It is important to note that concast does not require modifications to the forwarding algorithm of routers, beyond those already required for other protocols that do hop-by-hop processing, such as RSVP. Note also that concast is completely independent of multicast; concast-capable routers need not support multicast forwarding.

### C. Concast Signaling Protocol

The *concast signaling protocol* (CSP) deals with the establishment of flow state in the concast-capable nodes of the network. Here we merely outline the basic CSP operation to show its simplicity; see [12] for further details.

Hosts and routers in the network may vary in their level of support for concast. The lowest level is *no support.* These legacy nodes have no knowledge of concast datagrams, and do not implement the concast signaling protocol. Hosts that do not support concast can neither send nor receive concast datagrams. Routers that do not support concast will forward concast and CSP datagrams as regular unicast IP datagrams.

Nodes that are *fully concast-capable* support the merge framework as described above and carry out the merge specification of each active flow. In addition, fully concast-capable nodes run a CSP Daemon (CSPD) implementing the Concast Signaling Protocol to manage flow state and distribute merge specifications. Only fully concast-capable hosts can be *receivers.* A middle level of support is *sender-only.* Hosts that implement the concast signaling protocol but not the merge framework can function as concast senders but not receivers.

The signaling protocol distinguishes between concast

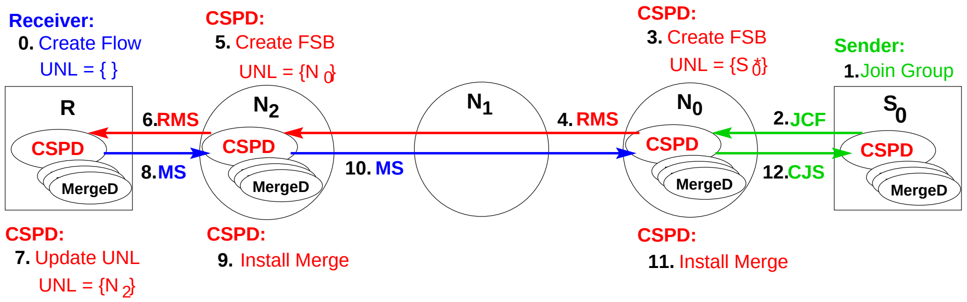


Fig. 2. Concast Signaling Events.

senders and concast-capable routers. Senders are generally leaf nodes in the concast tree; they do not have direct access to the merge specification, and thus cannot extend the tree. This separation of capabilities makes it possible for routers to apply access controls on membership in the concast flow.

All CSP messages are sent as regular IP unicast datagrams with the IP protocol number for CSP. Messages traveling downstream (toward the receiver) carry the Router Alert IP option to stimulate hop-by-hop processing. The source address in every CSP message is the unicast address of the last concast-capable node to forward the packet. In other words, the source address of a CSP packet is rewritten at every concast-capable hop. This is crucial because it enables connectivity problems in the concast tree to be detected via the Internet Control Message Protocol (ICMP) [14].

The normal sequence of signaling events is depicted in Figure 2. When the receiving application program signals its local concast module to create the concast flow, it supplies a merge specification. The local concast module creates a flow state block for the flow with an empty Upstream Neighbor List, checks the merge specification for well-formedness and conformance with local policy, then installs the merge specification and allocates computing resources for the flow. It then waits for senders to join the group.

The sending application at $S_0$ must signal its local concast module to join the group. $S_0$'s local concast module creates a FSB for the flow and sends a *Join Concast Flow* (JCF) message with source $= S_0$, destination $= R$, and containing the flow specification $(R, G)$. The JCF message travels toward the receiver $R$ until it encounters a concast-capable node (which may be the receiver — recall that the receiver must be fully concast-capable). The concast-capable node (in the figure, $N_0$) recognizes the CSP message and passes it to its local CSPD. If no FSB exists for $(R, G)$ at $N_0$, as depicted in the Figure, its CSPD creates one and initializes the UNL to $\{S_0^*\}$, where '$*$' indicates that the element is a **sender** (and not a router). Then $N_0$ sends a *Request for Merge Specification* (RMS) message with source $= N_0$, destination $= R$, and containing $(R, G)$.[2] The RMS message travels toward the receiver, with a new FSB being created at each hop, until it reaches a node that already has a Flow State Block for the flow; in the Figure this is the receiver $R$. Note that node $N_1$ does not support concast, and forwards the RMS message transparently. When $N_2$ creates the UNL for the $(R, G)$ flow, it places $N_0$, the source address of the RMS message, in it.

Node $R$ creates a *Merge Specification* (MS) message containing the code for the merge specification, and sends it with source $= R$ and destination $= N_2$. When the CSPD at $N_2$ receives the MS message, it installs the merge specification in the FSB and sends a MS message (with source $= N_2$) containing the merge specification to all members of the flow's UNL that are not marked sender-only, and a *Concast Join Success* (CJS) message to those upstream neighbors marked sender-only.

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 an MS message that propagates back upstream toward the senders via the Upstream Neighbor Lists. New senders are thus "grafted" onto the concast tree at the point where their path to the receiver intersects the existing tree.

Flow state is maintained using standard soft-state techniques. Each node periodically retransmits a "refresh" signaling message downstream on each flow to refresh its entry in the next hop's UNL for the flow, and waits to receive the response. UNL entries age and are removed if not refreshed within a "keepalive" period. If a flow's UNL becomes empty, the FSB is deallocated.

Errors are handled by sending CSP *Error* messages upstream. For example, if an RMS message reaches the receiver's node and the flow does not exist there, an Error message indicating "No Such Flow" is propagated back upstream to the originating sender. Such messages can also be used to quickly terminate a flow from the receiving host. Routing changes are detected transparently by the periodic retransmission of the CSP messages as part of soft-state refresh.

[2] Here $N_0$ is the address of the interface through which the RMS message is sent toward the receiver, and similarly for the other nodes in the Figure.

### D. A Prototype Implementation

We have implemented a complete prototype of the concast service on the Linux operating system platform. The implementation has both user-space and kernel-level components. The kernel components include:

• Socket options for *creating* a concast flow with an associated merge specification at the receiver, and for *joining* a concast flow as a sender.

• Modification of the Internet Protocol routines to recognize incoming concast datagrams and CSP messages by the presence of router alert and either the Concast Group IP Option or the CSP protocol type, respectively. We also added the necessary plumbing (intra-kernel routing) so that concast and CSP packets are diverted to/from the Merge and CSP Daemons (see below) for processing. Figure 3 shows the paths through the Linux kernel taken by concast packets that are sent, received, or forwarded by a node.[3]

• Control/Processing Endpoints are special sockets that support user-space *Merge Daemons* and a user-space *CSP Daemon*. Each node has a single CSP Daemon that listens on the *CSP signaling socket* for CSP messages. For each new flow the CSP Daemon creates a *merge socket* where concast datagrams belonging to the flow will be delivered during the forwarding process. The CSP Daemon then creates a Merge Daemon, which processes the the datagrams delivered to that socket, forwarding when *done*() becomes true.

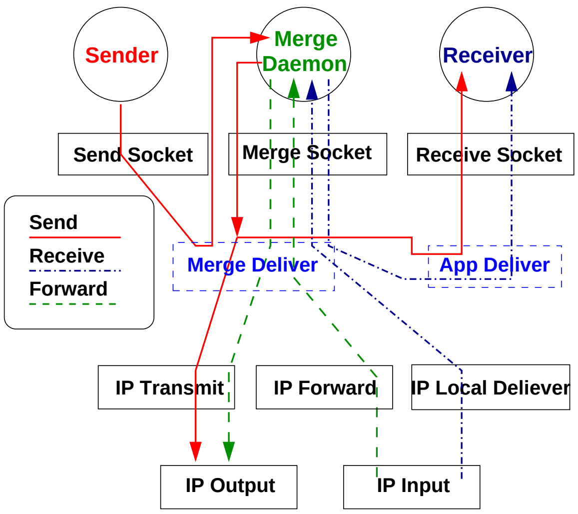


Fig. 3. Intra-node routing of concast packets in the prototype

The user-space components of our implementation include the *CSP daemon* and *Merge Daemons*. The CSP Daemon deals with CSP messages and manages per-flow state. Merge Daemons execute Merge Specifications. Our prototype currently supports two types of Merge Daemon: one that interprets merge specifications coded in Java bytecodes and another that interprets TCL scripts. We have tested the system with three merge specifications. One performs duplicate suppression, the other two perform audio and video merging. The audio and video merge functions combine digital streams sent to the same receiver into a single composite stream, thereby saving network bandwidth.

[3] We actually implemented an earlier version of the signaling protocol, which differs from that described in the previous subsection in that it does not distinguish between end systems and routers, so the merge specification is installed in the sending hosts. Thus, sent concast messages are shown being diverted to the Merge Daemon in the Figure.

### E. Compatibility Issues

Before presenting an example merge specification, we consider some minor details affecting the use and implementation of concast.

The first is *fragmentation*. If fragmentation occurs en route, the *getTag* function may become confused, causing a datagram fragment to be assigned to the wrong class. Probably the best solution for this problem is to employ path MTU discovery and have the sending application not send datagrams smaller than the Maximum Transmission Unit along the path to the receiver. For this to work, merge specifications also need to ensure that merged datagrams are no larger than the incoming datagrams from which they are formed. This is generally not a problem in multicast feedback applications, where payloads are typically small. Another possibility is for the application to add its own fragmentation and reassembly functionality to the merge specification.

Another issue is how to deal with *control messages*. The Internet Control Message Protocol (ICMP) [14] is designed around the notion that error (control) messages are sent to the source of the offending datagram. The source address in a concast datagram identifies the last concast-capable hop that forwarded it. When a concast-capable node receives an ICMP message referring to a concast datagram it forwarded, it may need to propagate information upstream, via the Upstream Neighbor List of the flow in question. Some intelligence may be required on the part of the router to determine whether or not the message should be propagated. For example, a Host Redirect message sent in response to a concast datagram would not need to be propagated, whereas a Destination Unreachable message definitely should be.

## V. A Scalable Feedback Service

In this section we describe a generic, scalable data-collection service based on the framework described above. The service solves the problem of applying an operation (such as conjunction or minimum) to information distributed across a group of nodes, and delivering the result to a particular node, the receiver. The size of the group is unknown to the receiver and to the group members. A need for this kind of service often arises in multicast applications, where the sender needs to know whether *some* group member failed to receive a message, or what the *maximum* loss rate, or *average* delay is. Use of unicast to send the values to the receiver and then compute the value results in implosion.

Let $G$ denote the group, and assume each group member $m$ has a value $v_m$, and let $\oplus$ be any associative and commutative operator. To use the service, a stimulus causes each group member $m$ to send a concast message containing a particular sequence number and $v_m$ to the receiver. For multicast applications, the stimulus will typically be a multicast message sent from the source, but it could also be timer-based. Eventually a message containing the result of applying $\oplus$ to all $v_m$ such that $m \in G$ is delivered to the receiver.

After giving the merge specification for our feedback collection service, we consider the problem of lost messages, and then present results of an experiment evaluating the effectiveness of our reliability solution and comparing it to traditional unicast solutions.

### A. The Merge Specification

The service operates in the expected way: at each concast node, the first value to arrive with a given sequence number is stored. Thereafter, when a message arrives, the stored value is replaced by the result of applying $\oplus$ to it and the value in the message. When a value has been received from each upstream neighbor, the stored result is placed in a concast message and forwarded toward the receiver. For the moment we ignore the problem of lost messages.

Each concast packet $p$ contains the following fields:

| | |
|---|---|
| *p.sender* : | ID (address) of sender |
| *p.seq* : | tag |
| *p.value* : | value |

The Merge State Block $s$ for this datagram equivalence class contains the following information:

| | |
|---|---|
| *s.seq*: | sequence number for this collection instance |
| *s.value*: | result of applying $\oplus$ to values received so far |
| *s.rcvd*: | set of neighbors heard from so far |

The functions making up the merge specification are informally defined as follows:

```
getTag(p)
   return p.seq.
done(s)
   return (s.rcvd = s.UNL)
bldmsg(s)
   return (p,null) where:
   p.sender = s.myID
   p.value = s.value
   p.seq = s.seq.

merge(s,p,f)
   if (s is NULL) then
      create a new merge state s;
      s.value = p.value;
      s.rcvd = {p.sender};
      s.seq = p.seq.
   Otherwise
      s.value = p.value ⊕ s.value;
      s.rcvd = s.rcvd ∪ {p.sender}
```

Here the variable $s.myID$ is assumed to contain the address by which the processing node is known to its downstream neighbors as an upstream neighbor, and $s.UNL$ contains the flows UNL.

Given that senders initially transmit their own values, it is not difficult to show that any message arriving at a node contains the result of applying $\oplus$ to all values originating in the subtree rooted at $n$. Note that for simple operators (e.g. min, sum), each of these functions requires a strictly bounded amount of time and space to compute; thus implementing them in hardware would be straightforward for any given operator $\oplus$ and type of value.

### B. Reliable Collection

For the merge specification above, the loss of a message at any point will prevent the delivery of the value to the receiver: Some node will fail to receive a message from one of its upstream neighbors, and will not forward its result; every concast node downstream from that node (including the receiver) will fail to hear from some upstream neighbor, and no final result will be delivered to the application.

Define a *concast link* as the downstream path between two concast nodes that are neighbors in a concast flow tree. If there are $L$ such links in the concast tree, and each link loses a packet with probability $p$, the probability that a result will be delivered is $(1-p)^L$, assuming losses are independent. The number of concast links is at least the group size, so as the group size grows, it becomes less and less likely that a result will be received. With 200 links and a loss probability of 1%, the probability of successful collection is less than 14%.

The brute force way to recover from loss is to reissue the stimulus repeatedly until the result reaches the receiver. The expected number of iterations is then $1/(1-p)^L$, about 7.5 for the 200-link example above. Thus a brute force approach to reliability has limited scalability. (Note that the likelihood of success using unicast collection is in general even smaller, because more links traversals have to occur without loss.)

To achieve a more robust collection service, we can modify the merge specification slightly so that for each incoming message the *merge*() function checks to see whether the sender of the message has already been heard from before processing the message. In addition, the message and state block include a subsequence number that records the number of times a value has been forwarded. Once a node has heard from all of its upstream neighbors, the *done* function returns true exactly once per subsequence number. With this modification the same sequence number can be re-used, and with each iteration, the values propagate closer to the receiver.

Next we compare the effectiveness of this reliable data-collection service to that of the traditional unicast feedback method.

### C. Simulation Analysis: Reliable Feedback Collection

To measure the effectiveness of our feedback service, we simulated a simple generic application in which a single receiver stimulates all group members (senders) to send it their current value. Our goal here is to evaluate the scalability of the service and compare its behavior under more or less controlled condition to a more traditional unicast approach.

We implemented two solutions, one using concast and the other unicast. In each case, the receiver stimulates the group, and in response all group members (senders) transmit their values to the receiver. If the result cannot be computed because one or more messages are lost on the way to the receiver, the receiver times out and restimulates the group and all nodes retransmit their values. Timeouts and retransmissions continue to occur until the receiver obtains the result. The whole process is repeated 100 times.

The concast solution is the reliable version described in the Section V-B. In the unicast solution, all senders unicast feedback packets to the receiver. A key advantage of concast is the fact that the collector does not need to know the members of the sender group or the group size. The unicast approach, on the other hand, cannot be implemented without this information; therefore in our simulation the receiver remembers the value received from each sender. Unicast collection succeeds as soon as a value has been received at least once from each group member. In practice this approach would have limited scalability because of the need to keep track of individual group members.

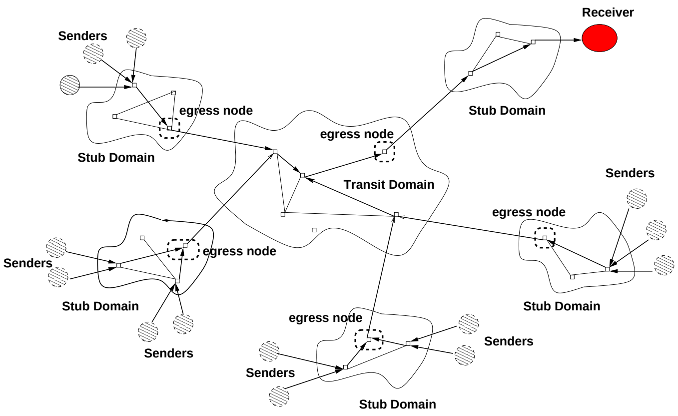


Fig. 4. Example of a transit-stub topology

Our simulations were run on the VINT/UCB/LBL Network Simulator (NSv2.1) [15] using transit-stub topologies generated by the GT-ITM topology generator( [16]). Transit-stub topologies identify some domains as transit domains, carrying traffic on behalf of other domains, and others as stub domains, which connect only to transit domains. Figure 4 illustrates the basic arrangement of a transit-stub graph. Our simulated topology had one "core" transit domain consisting of four nodes. Each node in the core has six stub domains attached. Each of the 24 stub domains has, on average, nine nodes, resulting in a total graph size of 220 nodes. The (concast) receiver and all senders are randomly placed in stub domains. The number of senders ranged from 50 to 180. Links between the core nodes have bandwidths of 100 Mbps. All links within a stub domain have a bandwidth of 10 Mbps. Instead of generating cross traffic to simulate packet loss, we used a uniformly distributed loss rate of 1% across all links. All feedback packets were 100 bytes long.

The graph in Figure 5 illustrates concast's scalability. It shows the number of stimuli required to deliver the result to the receiver 100 times. For small numbers of senders, unicast and concast exhibit similar performance. However, once the group size exceeds 120 senders, congestion occurs in the unicast application causing an exponential increase in the number of retransmissions. The problem is that every group member responds to the multicasted stimulus. Consequently, some members' packets are lost almost every time. Concast, on the other hand, requires only one message to be sent over any given link per stimulus, thereby avoiding self-congestion.

Figure 5 also includes an "egress-only" curve that illustrates performance when concast is deployed only at egress routers (see Section VII). The egress-only line shows that most of the benefits of concast can be obtained even when concast is only partly supported.

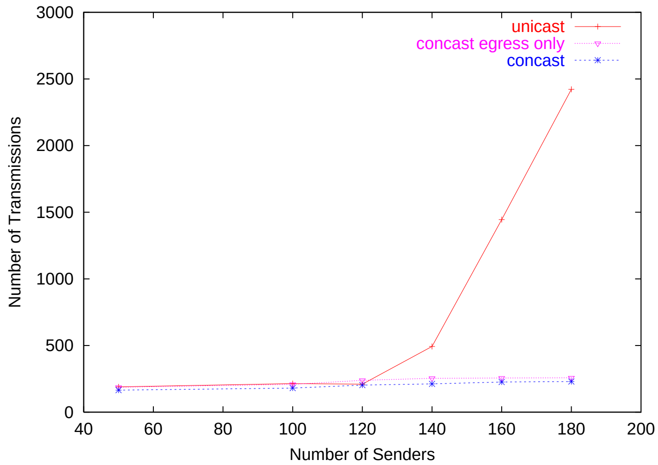


Fig. 5. Attempts required to collect 100 values

An alternative to waiting until all group members are heard from is to use the results obtained after some number of attempts. This produces an "approximate" answer, in that some senders' values are not included in the computation. One way to quantify the effectiveness of concast is to compare the accuracy of the approximate results it produces with those produced using unicast. To compare accuracy, we modified the feedback application so that in each trial, the unicast solution runs until the concast solution produces a complete result; at that point the value is computed. In Figure 6 we plot the average value of $\frac{Obtained\_Result}{Desired\_Result}$ for 100 trials at each group size. (The concast value is always 100%.) Again, unicast accuracy deteriorates quickly around 120 senders. Although this type of inaccuracy may be tolerable for certain applications, it still consumes unnecessary bandwidth, causes congestion, and requires the receiver to know about all senders.

## VI. Limiting Resource Consumption

If concast (or any active network service) is to be deployed, it must be shown to pose no threat to existing services nor to the users of the network. For example, the presence of concast in the network should not make it any easier to mount a denial-of-service attack. Also, there must be some way to ensure that users will receive adequate ser-

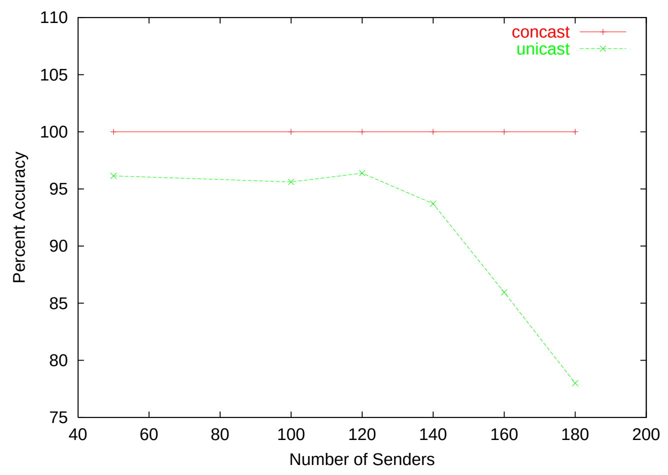


Fig. 6. Concast vs. Unicast Accuracy of the SUM operation.

vice even under congested conditions, i.e. when demand exceeds supply. Finally, it must be possible to bound the amount of resources — computation, bandwidth, and state storage — used at any node by any particular concast flow.

Routers supporting concast must maintain state per concast flow. The amount of state per flow can be limited by the processing framework in the manner described earlier (Section III). That is, the framework permits only a fixed number (say $N$) of datagram equivalence classes to be active at any time, and also limits the size of the merge state block stored for each DEC. The constraint on number of active DECs is silently enforced by discarding an old merge state block when an arriving datagram belongs to a new DEC and $N$ DECs are already active. This is reasonable because the service is best effort; it is the responsibility of the application to ensure that no more than $N$ DECs are active at any time.

Another issue is computing bandwidth: Applying a merge specification requires processor cycles. Modern routers have highly optimized processing paths for forwarding normal IP datagrams. Concast processing occurs off the fast path and thus does not interfere with normal traffic passing through the router. However, "slow path" capacity is generally limited, and is shared among a variety of control functions in a typical router. We expect that concast-capable routers would possess extra computing and storage capacity (possibly dedicated to merge processing), and provide isolation between merge processing and router control functions.

The amount of computing bandwidth used in merging packets for one flow can be limited through straightforward use of watchdog timers or similar techniques. An alternative approach (which we have not pursued) would be to require merge specifications to be supplied in a programming language that allows a bound on compute time to be verified when the merge specification is first installed.

The concast service uses link bandwidth to (1) transmit CSP messages and (2) forward concast data packets. Bandwidth consumption of the CSP protocol is similar to that of any other control protocol based on soft state. For data forwarding, the merging framework guarantees that the time-to-live field of every outgoing datagram is smaller than the TTL of every incoming datagram for that flow, so each concast packet consumes a bounded amount of network bandwidth. A bigger concern is that the hop-by-hop merge processing could be used to "amplify" a flow, thus providing a tool for denial-of-service attacks. As defined in Figure 1, the merge framework produces *at most* one output packet per invocation (i.e. per incoming packet), so this is not a problem. However, if the merge function is allowed to set callbacks (timeouts), this no longer holds. Even with only one pending timeout allowed at any time, as many as two outgoing packets per incoming packet could be produced at a node. This means that exponential growth of a packet flow along a path is at least theoretically possible.

To eliminate this potential doubling effect, the merge framework must cancel any pending timeout whenever the *buildMsg*() function is invoked. As long as timeouts can only be set in the *merge*() function, the cancelled timeout cannot be restarted until a subsequent packet arrives. As a result, the number of outgoing packets cannot exceed the number of incoming packets.

As for the number of concast flows that can be handled at any node, each concast flow is associated with an $(R, G)$ pair. This is comparable to current shortest-path multicast implementations, which keep state for each $(S, G)$ flow, i.e. multicast source $S$ sending to the group $G$. Thus in terms of per-flow state requirements concast scales like shortest-path or single-source multicast.

Finally, it is important to note that the signaling requirements of concast allow for admission control: a node that is overloaded can simply refuse a request to become part of a concast flow. This provides a mechanism to ensure that existing flows receive adequate service.

## VII. Partial Deployment

It is unreasonable to assume that every router in the Internet will instantaneously support concast (or any new service for that matter). Moreover, because concast requires per-flow state, it may not be suitable for deployment at backbone nodes supporting hundreds of thousands of simultaneous packet flows. It is therefore important that concast provide demonstrable benefits to applications even when partially deployed.

In the extreme case, only the end-systems (senders and receiver) support concast, with no intermediate nodes supporting it. Although such a configuration provides no savings in network bandwidth, and does not preclude implosion at or near the receiver, it does shield the application from details of the number and identity of senders.

On the other hand, deploying concast at strategic locations in the network provides substantial benefits, while minimizing the number of concast-capable nodes required. The question that arises is whether a small number of concast-capable routers can achieve comparable performance benefits to the case where all nodes support concast (i.e., concast everywhere).

It is generally true that resources are most scarce around the edges of the network. Bottlenecks typically occur at links that cross routing domains boundaries. Therefore it makes sense to apply concast processing at either end of such links. Applying concast processing at the "inbound" node is too late: the scarce resource has already been consumed. On the other hand, a domain's *egress nodes* lie at the head of a potential bottleneck link, and thus are strategic locations for the deployment of concast capabilities, reducing the bandwith across the bottleneck link and avoiding implosion. Thus we expect that most of concast's benefits could be obtained if it is deployed at all egress nodes.

To quantify the benefits of partial deployment at egress routers, we evaluated the performance of concast in *unicast* (no concast routers), *concast-everywhere* (concast on all routers), and *concast-egress* (concast only at egress routers) scenarios. To allow for the effect topology has on each deployment scenario, we developed four distinct topology types and evaluated each deployment scenario for ten graphs in each class. The network topologies were modeled as *transit-stub graphs* [16] with several *transit* domains serving as the core of the network, interconnecting many *stub* domains. We varied the size and quantity of transit and stub domains, creating topological graphs that ranged from a few large core (transit) domains connecting many stub domains to topologies with many small transit domains connecting a smaller number of stub domains. We found that for these four types the network topology does not significantly influence the benefits acheived by a concast service. Consequently we show only the results for one topology type; graphs for the others are similar.

The results presented are for graphs with 20 transit domains, 7 nodes (mean) per transit domain, 7 stub domains (mean) per transit node, and 5 nodes (mean) per stub domain, for a total graph size of 4,900 nodes. Senders were randomly selected from stub domains as was the receiver.

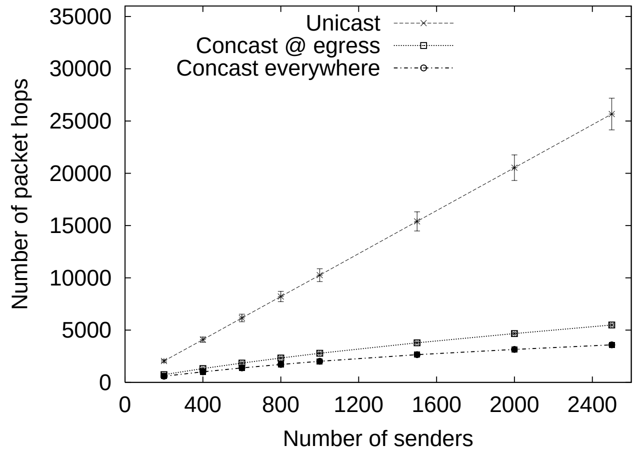


Fig. 7. Packets hops needed for unicast, concast-everywhere, and concast-egress.

To compare the three deployment scenarios, we consider two simple metrics on these graphs: *packet-hops* and *fan-in*. *Packet-hops* refers to the number of times a packet is sent over any link (i.e., summing the number of links in the tree, with a link counting once for each "child" reached via it).

Figure 7 shows the average number of packet-hops needed as the number of senders varies from 200 to 2500. The figure plots the mean value for the 10 sample graphs as well as the 95% confidence interval.

As expected, packet-hop counts for concast-everywhere are significantly lower than the packet-hop counts for a unicast deployment model. Intuitively, each sender in the unicast model contributes (to the total hop count) its own distance to the receiver. Thus the hop count is roughly $\mathit{NumberOfSenders} \times \mathit{AverageSenderDistance}$, which explains the linear increase shown in the figure. On the other hand, concast-everywhere senders only contribute the cost to reach the tree (as opposed to the root). On average, the distance to reach the tree is only a fraction of the cost of reaching the root. As the tree size increases, the number of additional packet-hops per additional sender begins to approach one (it can go below one if some senders lie on the path of other senders).

Figure 7 also confirm ours hypothesis that concast-egress achieves most of the advantages of concast-everywhere with only partial deployment. On average, concast-egress attains 89% of the benefit seen by concast-everywhere.

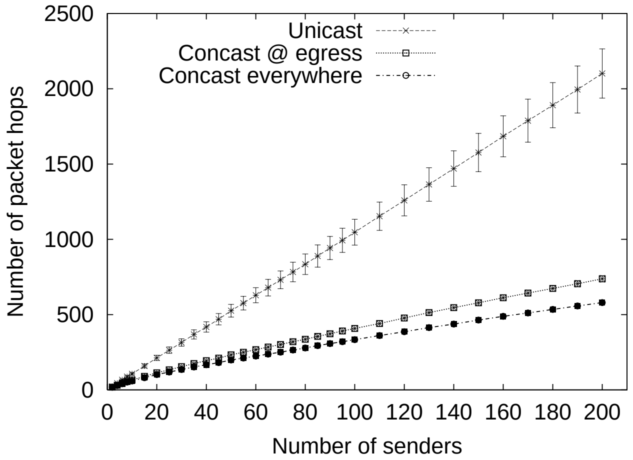


Fig. 8. Packets hops needed for smaller numbers of senders.

Moreover, Figure 8 (a more detailed view showing smaller numbers of senders) indicates that the benefits of concast-egress and concast-everywhere are realized even when the group size is relatively small. With as few as 10 group members, the new senders have a high probability of being combined with existing senders, reducing the packet-hop count.

Table I shows the variation across topologies by comparing the minimum and maximum slope of the packet-hops line for the 10 sample graphs. It is interesting that unicast has a much higher variance than both concast-everywhere and concast-egress. This higher variance results because even small changes in the topology (particularly near the receiver) can produce large changes in the shape of the tree.

TABLE I
PACKET-HOPS PER SENDER, AVERAGE OF 10 GRAPHS.

| Approach | Min | Max |
|---|---|---|
| Unicast | 9.66 | 10.87 |
| Concast-Everywhere | 1.30 | 1.30 |
| Concast-Egress | 2.03 | 2.10 |

A key factor in concast's scalability is *fan-in*: the number of upstream neighbors a concast node has. Obviously, universal support for concast results in smaller fan-in than partial support. To illustrate, we measured the fan-in at each concast node as the number of senders increased from 200 to 2500. Figure 9 plots the maximum and average fan-in values for concast-everywhere and concast-egress. (For unicast, fan-in is equal to the number of senders.)

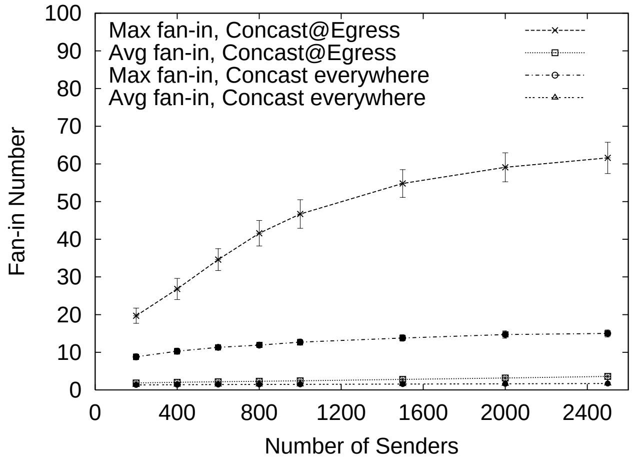


Fig. 9. Maximum and Average Fan-in.

In addition to these simple graph measurements, we measured the performance of the feedback-collection service described in Section V, when concast was deployed only at egress nodes. The "concast egress only" curve in Figure 5 shows performance very close to that of the concast-everywhere curve. (Note: the networks used in Figure 5 are much smaller than the ones considered in this section.)

Our analysis shows that deploying concast only at domain egress nodes provides almost the same benefits as concast-everywhere, while keeping fan-in at reasonable levels.

## VIII. Conclusions

We have presented concast, a many-to-one network service that is in many ways symmetric with multicast. Concast offers a solution to a problem arising in many networked group communication applications: how to collect feedback while avoiding implosion. It is especially useful in the context of reliable multicast, but does not rely on multicast in any way for implementation.

Concast can be implemented in a number of ways; in its most general form it combines *application-independent* network support with programmability to create an active Internet service. However, modifying the behavior of the Internet infrastructure in this way is a challenging proposition. New network-layer services of any kind—programmable or otherwise—are unlikely to be deployed unless they meet certain conditions.

First, a new network-layer service must solve a significant problem, one that cannot be satisfactorily solved—for example, in a scalable manner—without assistance from the network. Concast solves real problems that have proven difficult to solve without network support. Our initial results (Section V-C) are consistent with the claim that concast can be used to collect feedback in large-scale multicast applications.

A second requirement for deployability is backward-compatibility with existing network mechanisms and protocols. Concast requires no new routing or other algorithms, and does not break any existing Internet service. Concast data and signaling packets are handled as normal unicast packets by legacy routers.

Third, it should be acceptable for the service to be implemented "off the fast path". The forwarding computation in modern routers is highly optimized for speed. Any new service that modifies that computation is probably incompatible with high-performance, forwarding-intensive applications. Moreover, routers have limited computing and memory resources, and a new service's requirements should be modest. Concast packets are diverted off the fast-path and thus do not interfere with standard fast-path processing of IP datagrams. This limits its applicability for high-performance applications, but a significant number of the applications we have found either involve control plane data or have otherwise-limited performance requirements. Moreover, we have described (Section VI) how the resources consumed by any particular concast flow can be bounded via the merging framework.

Finally, limited or partial deployment of the service must produce immediate and tangible benefits. If the entire end-to-end path between users must be upgraded in order to provide value to applications, it may take a long time before the service is widely available and used (cf. integrated services). Moreover, other factors besides the impracticality of "flag day" upgrades may prevent universal deployment. For example, per-flow state and computational requirements may preclude its deployment in the core of the network where hundreds of thousands of flows pass through each node; policy considerations may also make it difficult to support a service in certain parts of the network. An example is the reluctance of some ISPs to enable Internet multicast service. Because concast is backward compatible, it can be partially deployed in the Internet; we have shown (Section VII) that most of the benefit of concast can be achieved simply by deploying concast at egress routers.

These are *necessary* conditions, and not sufficient ones. Ultimately, deployment of a new service has to be justified on economic grounds—which boils down to how useful the service is to paying customers, and how much it costs to provide it. As part of our ongoing work we are adding support for group admission control, so that concast can be offered as a form of premium service.

Concast solves a real problem that has defied end-system-only solutions. The varying needs of applications with respect to merge semantics and the timing of delivered messages makes it difficult to justify implementing a single set of semantics within the network. However, the ability to place application-specific processing just where it is needed in the network by downloading code makes this a much more attractive proposition. Indeed, concast seems to be an ideal match for active networks, and could provide substantial motivation for the deployment of active network capabilities in the infrastructure.

## References

[1] Van Jacobson and Steve McCanne, "WB: The whiteboard application," ftp://ftp.ee.lbl.gov/conferencing/wb/.

[2] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, "Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," in *SIGCOMM*, Cambridge, Massachusetts, September 1995.

[3] DataBeam Cooperation, "The LearningSpace$^{TM}$ Server," http://www.lotus.com/home.nsf/welcome/learnspace.

[4] The Microsoft Cooperation, "The Microsoft Netmeeting Program," http://www.microsoft.com/windows/netmeeting/.

[5] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper, "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, January 1998, http://www.uk.research.att.com/vnc/.

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

[7] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele, and E. Zegura, "Active Reliable Multicast on CANEs: A Case Study," April 2001.

[8] S. Kasera, S. Bhattacharjee, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele, "Scalable Fair Reliable Multicast using Active Services," *IEEE Network Magazine (Special Issue on Multicast*, January/February 2000.

[9] J. Rosenberg and H. Schulzrinne, "Timer Reconsideration for Enhanced RTP Scalability," in *IEEE INFOCOM*, March 1998.

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

[11] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments In System Design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277–288, November 1984.

[12] Ken Calvert, James Griffioen, Amit Sehgal, and Su Wen, "Concast: Design and Implementation of a New Network Service," in *Proceedings of 1999 International Conference on Network Protocols*, November 1999.

[13] D. Katz, "IP Router Alert Option," February 1997, Internet Request For Comments 2113.

[14] J. Postel, "Internet Control Message Protocol," September 1981, RFC 792.

[15] The MASH Research Team, "The ns network simulator," http://www.isi.edu/nsnam/ns/.

[16] Ellen W. Zegura, Kenneth Calvert, and S. Bhattacharjee, "How to Model an Internentwork," in *IEEE INFOCOM'98*, San Francisco, CA, April 1998.

**Kenneth L. Calvert** is an Associate Professor in the Department of Computer Science at the University of Kentucky, where he does research on the design and implementation of flexible network services. His current interests include active networks, secure multicast, and abstract models of network topology. Dr. Calvert holds a Ph.D. from the University of Texas at Austin (1991), a M.S. from Stanford University (1980) and a Bachelor's degree from M.I.T. (1979). From 1979 to 1984 he was with Bell Telephone Laboratories, and from 1991 to 1998 he was with the College of Computing at Georgia Tech.

**James Griffioen** is an Associate Professor of computer science and the Director of the Laboratory for Advanced Networking at the University of Kentucky. His research interests include network protocol design, active network services, distributed file systems and operating systems.

**Su Wen** received her B.S. degree from Bulter University at Indianapolis, IN, and is currently pursuing a Ph.D. degree in Computer Science at the University of Kentucky. Her research interests include Active Networking, multicast routing protocols, and flow, congestion control and QoS for multimedia data delivery.

**Amit Sehgal** Amit Sehgal completed his Bachelors in Computer Engineering from Bombay University, India, and is now working towards his Ph.D. at the University of Kentucky. He is interested in Active Networking, Multicast and Security protocols.

**Billy C. Mullins** received a B.A. degree in computer science from the University of Kentucky in 1998. He is currently pursuing the Ph.D. degree in the Computer Science Department at the University of Kentucky under the guidance of Ken Calvert and Jim Griffioen. His research interests include network technologies, and operating systems.