Implementing a Concast Service*

K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, S. Wen
[calvert,griff,mullins,asehgal,suwen]@dcs.uky.edu
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046

Abstract

Concast is a new network service, the dual of multicast. With concast, messages from different senders addressed to the same receiver are merged by the network into a single message for delivery to that receiver. Like multicast, concast is a scaling mechanism that simplifies application design by allowing a group of peers to be represented by a single (source) address; it is useful in dealing with implosion problems that can arise when many receivers have to provide feedback to a sender. Like multicast, concast conserves network resources by reducing redundant packet transmissions. Unlike multicast, concast is most useful when the application can customize the service by supplying the definition of the "merge" function to be implemented by the network; it is thus an excellent match for active networking. This paper describes a concast framework we have implemented. In spite of its programmability, the implementation is compatible with current Internet protocols, and could be deployed incrementally.

1 Introduction

The Internet is rapidly becoming the virtual workplace where people go to meet, share information, discuss ideas, and, in general, collaborate. Collaborative applications such as teleconferencing, virtual-offices, shared whiteboards, chat rooms, and application sharing are becoming important tools. Information dissemination applications such as stock price updates, network news feeds, mailing lists, and webcasting to interested parties has become the lifeline of many businesses. As a result, flexible group communication models are becoming increasingly important. Although group communication models have been in use for many years [6, 4, 7], past approaches have focused on the problem of one-to-many communication, with extensions to many-to-many. A glaring omission is the lack of network support for many-to-one communication.

We have proposed a new programmable network service called *concast*, which is an inverse or dual of multicast [5]. Multicast uses a single network address to represent a group of

*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.

receivers; concast uses a single address to represent a group of *senders*. When a packet is sent to a multicast address, the network delivers a copy to all receivers who have joined that group. When multiple members of a concast group send packets addressed to a single receiver, only a single packet is delivered to that receiver, and it appears to come from the concast group. Internet multicast is a scalable one-to-many service because the sender need not deal with receivers individually, but instead can treat them as a group. Concast offers similar benefits to applications that require scalable many-to-one communication. Examples of such applications include sensor data collection, group teleconferencing, and any application involving feedback from a multicast group.

A key difference between concast and multicast is the requirement of, and benefit from, having a *programmable* concast service, where applications can *customize the merge semantics* carried out by the network. Applications that can download code into the network to map from the set of sent messages to received message means that any application involving many-to-one communication can potentially benefit from a concast service simply by tailoring the merge function to its needs.

This paper focuses on issues related to the implementation and deployment of a backward-compatible, customizable concast service, based on an early prototype using Linux PCs. Elsewhere we have defined the basic concast service framework and signaling protocol [5]. We begin with an overview of the service abstraction (Section 2) and then describe an implementation in the context of the IP protocol, highlighting problems that must be addressed in this context (Section 3). Section 4 presents the concast API and its implementation. Section 4.3 describes how merge processing can be added to Linux routers to convert them to concast-enabled routers. Finally, we illustrate the utility and customizability of the service with an audio mixer application (section 5).

2 Concast Service Overview

Concast is naturally symmetric with multicast in many ways. Like multicast, concast is a best-effort service, and requires that a portion of the IP address space be set aside for *concast group addresses*. In multicast, the sender does not know who the receivers are; in concast the receiver does not know who the senders are. When a packet is sent to a multicast address, the network delivers a copy to all receivers who have joined that group. The network *duplicates* multicast packets, so a single send results in multiple copies of the sent message being delivered; with concast, the network applies an application-specific, semantics-based *merge* function to map multiple sent messages into a single received message.

The concast receiver is responsible for providing the network with a *merge specification* that defines the mapping from sent messages to the delivered message. The merge specification includes a flow specification that describes the packets belonging to the flow, and descriptions of the various computations that must be carried out on the packets to merge them with other packets belonging to the flow. Each concast sender is responsible for signaling its intent to send as a member of the group (see Section 2.2). Signaling causes the merge specification to be pulled from the receiver toward the sender.

As concast datagrams travel through the network, they are processed hop-by-hop. At each concast-capable node, concast packets (flagged by the Router Alert IP option [9] and a concast address in the source field) are diverted for merge processing. Packets are classified into flows based on source (concast address), destination, and other application-supplied information (Section 3); the relevant merge specification is retrieved; and the flow-specific processing defined by the merge spec is applied.

```
void ProcessMessage(Receiver R, Group G, AppID A, IPdatagram m) {
 FlowStateBlock fsb;
 MsgKey k;
 MergeStateBlock *s;

  fsb = lookUpFlow(R,G,A);
  if (fsb != NULL) {
    t = fsb.getTag(m);
    s = fsb.findMergeState(t);
    s = fsb.merge(s,m,fsb);
    fsb.saveMergeState(s,t);    /* replace previous state value */
    if (fsb.done(s))
      forwardMsg(R,G,fsb.buildMsg(s));
  }
} /* ProcessMessage */
```

Figure 1: Network per-packet processing.

2.1 Concast Processing

Concast-capable nodes maintain a *Flow State Block* (FSB) for each active concast flow. The FSB contains the merge specification, processing state information for any in-process merges, and an *Upstream Neighbor List* (UNL) containing addresses of concast-capable nodes for which this node is the next downstream hop towards the receiver of the flow (R). The hop-by-hop processing of concast messages is defined in terms of the following functions, which together with the flow specification make up the *merge specification*:[1]

getTag(m)**:** a *tag extraction* function returning a hash or key identifying the message. Messages m and m' are eligible for merging iff *getTag*(m) = *getTag*(m').

merge(s, m, f)**:** the function that combines messages together. The first parameter is the current *merge state*, which contains information representing messages that have already been processed. The second parameter m is the incoming message, which will be merged into the saved state s. The third parameter is the flow state block for the concast flow to which m belongs.

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

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

The processing applied to each concast packet hop-by-hop is shown in Figure 1. When the node detects an incoming concast packet, it first retrieves any state associated with the given flow, which is identified by the receiver (destination IP address), group (concast address), and Application ID (carried in an IP option, see Section 3). If state exists, the identifier (tag) of the message being processed is computed using the receiver-supplied *getTag*() function, and any associated message state is retrieved. A new message state block is computed from the old state and the message payload, and the result is stored back with the tag. If the *done*() predicate indicates that merging is finished, the *buildMsg*() function is called to generate a payload, which is forwarded toward the receiver with the concast source address and the IP address of the receiver for this flow.

[1] We do not deal with the issue of a language for specification of these functions by the receiver, except to note that languages for specifying code in active networks are the subject of ongoing research [8, 13, 11].

2.2 Concast Signaling

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 [5] for further details.

CSP messages are sent as regular IP unicast datagrams with CSP identified in the protocol field, and the Router Alert IP option included to stimulate hop-by-hop processing. The signaling process begins when a receiving application indicates to the local concast module its desire to receive from a specific concast group, and supplies a merge specification. The local concast module creates a FSB for the flow with an empty UNL and installs the merge specification. On the other end of the flow, before sending a concast packet from group G to R, a concast sender first signals its intention to do so. This triggers the local concast module to check whether there already exists a FSB for (R,G). If not, the local module creates a FSB for the flow and sends a *Request for Merge Specification* (RMS) signaling message addressed to the receiver.[2] As the RMS message travels toward the receiver, it is intercepted by each concast-capable node along the way, until it reaches a node with the merge spec. Each node along the way creates an FSB, adds the source address to the UNL for the flow, inserts its own address as the source, and forwards the request.

When the message reaches a node that has the merge specification installed — in the worst case R — that node replies with a *Merge Specification* (MS) message containing the spec supplied by the receiver (encoded in a platform-independent way). Thus the merge specification propagates from the receiver to the senders. New senders are "grafted" onto the concast tree at the point where their path to the receiver intersects the existing tree (which may be at the receiver itself). Intermediate nodes that have only a single upstream neighbor do not invoke the merge function on incoming messages, but forward them immediately for efficiency.

Each concast-capable node periodically sends a "KeepAlive" message downstream for each of its active concast flows. If a node fails to receive a "KeepAlive" message from an upstream neighbor node for several periods in a row, it "prunes" that branch of that flow by removing the node from the flow's UNL; thus inactive senders are eventually removed from the tree. When a flow's UNL becomes empty, its 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 receiver has not provided a merge spec for the given flow, an Error message indicating "No Such Merge Spec" is propagated back upstream to the originating sender.

3 Issues

A number of issues arise in the design and implementation of a backward-compatible concast service. In this section we outline them and briefly describe the approach we have chosen.

Identifying Concast Flows. As soon as application-specified semantics are admitted, the possibility arises that different receiving applications on the same host will need different forms of the service and will supply different merge specifications. The network has to be able to distinguish among these flows so that the function `lookUpFlow` (Figure 1) returns the proper flow state block for each incoming datagram. At least two solutions are possible. One identifies

[2] The source address of a CSP packet is always the (unicast) IP address of the packet's origin; this is crucial because it enables connectivity problems in the concast tree to be detected via the Internet Control Message Protocol. (ICMP messages cannot be sent in response to packets with concast source addresses.)

a flow with the pair (receiver IP address, concast group); the other identifies it with the four-tuple (receiver IP address, receiver protocol, receiver port number, concast group). The first approach does not require the network to be aware of higher-level protocols, but requires that different applications on the same host use different concast addresses when they receive from the same concast group. The second approach allows the concast address to represent the group of senders, independent of receiver, but violates layering and relies on having port numbers in all transport protocols in the same place.

We decided to implement a third approach that borrows from the strengths of both. In our approach each concast service is identified by a tuple (G, R, A), where G is the concast group address, R is the receiver's IP address, and A is an ID that uniquely identifies the receiving application.

G and R are of course carried in the IP header; to avoid depending on application identification being in a particular location in the IP payload we place the application ID A in an IP option. The "AppID" option contains an opaque 32-bit number. How the AppID is chosen (and agreed upon between senders and receivers) is not defined by the concast mechanism; one simple way to ensure uniqueness is to derive the AppID from the receiver's protocol and transport address, which must be known to all parties anyway.

Flow State Scalability. Concast requires that nodes maintain state information for each flow (receiver and group). For scalability it is necessary to limit the per-flow state to a bounded (preferably fixed) size. Enforcement of this requirement is closely tied to the form of the programming interface. If merge specification functions are provided in the form of Java bytecodes, for example, the virtual machine used to execute those functions needs to ensure that their resource usage is limited. (Various approaches to this problem have been proposed in the context of active networks [13, 8].) For example, only a fixed number of in-progress merge states could be kept for any particular flow, with the size of each limited to a network-wide bound. (Note that this implies a limit on the rate at which the network can merge packets for any particular flow.) Given a bound on the amount of per-flow state, concast is approximately as scalable as multicast, which requires routing state for each (source, group) flow.

Fragmentation. So far our description has assumed that senders' IP packets were not fragmented. However, if the sender transmits large IP datagrams, routers along the way may split them into fragments, spreading the original payload across the fragments. In general, fragmentation is a problem because merge functions will typically be designed to operate on full packets.[3] Possible solutions include reassembling fragmented datagrams before handing them off for concast processing (which is likely unacceptable due to processing load, delay, etc); path MTU discovery by the senders; and having the application do its own fragmentation and reassembly, of which the merge function is aware. Our recommended approach is per-sender PMTU discovery.

4 Implementing the Concast Framework

To evaluate the benefits of the concast service model, we are implementing a prototype concast framework using Linux PCs as the development platform. In the following we describe

[3]Designing the *getTag* and *merge* functions to operate on fragmented datagrams requires that the merge operation commute with reassembly, that is, the result of reassembling the merged fragments is the same as the result of merging the reassembled fragments at each hop.

our design and illustrate where concast processing fits in the standard protocol stack. Our discussion is couched in terms of the Unix operating system, but a similar approach could be followed with most any operating system (particularly ones that support sockets). Our current prototype does not yet implement the CSP signalling protocol, and thus, merge functions must be installed manually at routers rather than being pulled down automatically. However, the signalling protocol described in section 2.2 can be implemented in our framework in a straightforward manner.

4.1 The Concast Programming Interface

Applications on the end-systems enable concast service by creating sockets in the normal fashion, and then associating the socket with a concast flow. Two new system calls are provided to associate a socket with a concast flow: `CCAST_JOIN_FLOW` and `CCAST_SET_MERGE`. `CCAST_JOIN_FLOW` is invoked by senders, while `CCAST_SET_MERGE` is invoked by receivers. Both calls are implemented as `setsockopt()` calls as shown in Figure 2.

```
setsockopt( sock, IPPROTO_IP, CCAST_JOIN_FLOW, &flowspec, sizeof(flowspec) )
setsockopt( sock, IPPROTO_IP, CCAST_SET_MERGE, &mergespec, sizeof(mergespec) )
```

```
struct flowspec {
  ulong ConcastAddress;
  ulong ReceiverIPAddress;
  ulong AppID;
  ushort ReceiverPort;
}
```

```
struct  mergespec {
  struct flowspec fs;
  char *getTag;
  char *merge;
  char *done;
  char *buildMsg;
}
```

Figure 2: The `CCAST_JOIN_FLOW` and `CCAST_SET_MERGE` system calls.

The `CCAST_JOIN_FLOW` call takes a flow specification and sets up the socket for concast communication. Specifically, it binds the socket to the specified source and destination addresses. All packets sent via this socket will have `ConcastAddress` as the source address and `ReceiverIPaddress` as the destination address; if the protocol supports port numbers, `ReceiverPort` will be used as the destination port number. Because the endpoints of the socket are fully specified, calls such as `sendto()` and `receivefrom()` are not permitted. In addition, the socket is configured so that the IP Router Alert and the IP AppID options will be placed in the IP header of each outgoing packet. The method of computing the AppID value (see section 3) is up to the application; in our audio application (see section 5) we set $AppID = (Protocol << 16) \mid ReceiverPort$. `CCAST_JOIN_FLOW` also initiates the CSP signalling protocol as described in Section 2.2. The call blocks until a Merge Specification or Error message (e.g. No Such Merge Spec, Destination Unreachable), is received for the flow.

The `CCAST_SET_MERGE` call binds the socket to a flowspec, but also installs the merge functions needed at concast-enabled routers. Specifically, it registers the socket to receive packets belonging to the flow (i.e., packets with `ConcastAddress` as source, destination IP address `ReceiverIPaddress`, and IP Application Id option with containing value `AppID`. (It also binds the receiver to the local port `ReceiverPort`.) It also causes the specifications for `gettag`, `merge`, `done` and `buildmsg` —here we assume they are supplied as character strings— to be parsed and compiled/installed locally so that they can be applied to incoming messages. The specifications themselves are stored with the FSB, for transmission in response to subsequent CSP Request for Merge Specification messages.

4.2 End-System Processing

The sender and receiver protocol stacks must be modified in order to transmit and receive concast packets. Figure 3 illustrates the protocol stack organization used at both the sender and receiver.

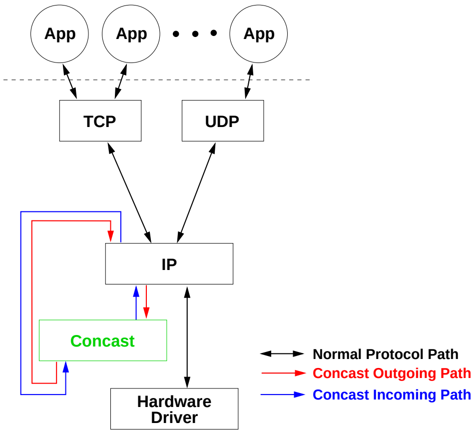

Figure 3: Sender/Receiver protocol stack organization

The concast module at the end system interfaces with the IP module, and is responsible for carrying out merge processing whenever the end system is a merge point for a flow (e.g. two senders on the host sending on the same flow, or incoming branches from different upstream neighbors destined for a local receiver). Flow identification is based on the flowspecs set via the `setsockopt()` calls. Outgoing concast packets are handed off to the concast module during IP processing for insertion of a concast source address, router alert option, AppID option, and merge processing if applicable (i.e. if the flow has other senders on the same host). Incoming concast packets are pulled off the normal IP processing path by the IP router alert option and passed to the concast module for merge processing. The concast module applies the merge functions specified in the `setsockopt()` calls, saving the merge state in a FSB until a merged packet can be given to the IP layer for delivery. When processing is finished, concast packets are then returned to the IP module for forwarding—either transmission on a network interface or delivery to a higher-level protocol. Similar layering organizations have been used by other protocols such as IPSEC [12, 10].

Our current implementation supports the `setsockopt()` calls required to bind the socket to the specified flowspec. Outgoing packets are marked with the IP Router Alert. Merging at the sender is not yet implemented, so only one (G,R,A)-bound socket per host is allowed. The merge function is applied at the receiver, but is currently applied at user-level inside the receiving application.

4.3 Concast Router Implementation

Merge processing at the concast routers (Linux machines) occurs in user-level processes, each executing in its own protected address space. Because we have not implemented the concast signalling protocol, merge functions must be manually started for the concast flow. Merge functions are currently written in C/C++ and linked into the concast daemon, which invokes them via calls to *getTag*(), *merge*(), *done*(), and *buildMsg*(). The daemon maintains a flow state block (FSB) for each active flow, storing the per-tag message state used by the *merge*()

and *buildMsg*() functions. To allow the routers to intercept concast packets, we modified the Linux kernel to direct packets with IP router alert and concast source addresses to a kernel module that demultiplexes the packet based on (G, R, A), where A is obtained from the IP AppID option.[4] Each concast daemon executes a flow-specific merge function. It opens a raw IP socket for receiving packets and issues a `setsockopt()` call that we have added to the Linux kernel which registers the daemon's interest in all concast packets (with Router Alert set) from flow (G, R, A). The *buildMsg*() function forwards packets using a socket that routes packets in the standard way (based on destination address).

5 An Audio Merge Function

A research area that has seen a flurry of activity in recent years is the problem of transcoding network flows [1, 2, 3]. The problem of transcoding arises in a wide range of contexts. However, the basic problem is the same: a flow of data from a sender to a receiver must be modified in some way at intermediate nodes in the network. Consequently, transcoding involves placing application-specific processing in the network to modify the flow of data. When multimedia flows converge in the network upstream of a link that has insufficient bandwidth for the load represented by the individual streams, a typical solution is to place a combiner at the point of convergence. Such a combiner, which takes in individual media streams and emits a single stream carrying the aggregated signal, is a natural application for concast. In the following we look at one such application, an audio combiner that takes in multiple μ-law-encoded (8000 8-bit samples/sec) streams and emits a single such stream, carrying the sum of the incoming signals.

We implemented an audio transcoding system of this type and measured its effectiveness using our concast framework. Our simple audio mixing function converts incoming μ-law streams to 14-bit linear samples, sums them, and converts the result back to a 64 Kbps μ-law stream; peaks that exceed the dynamic range are clipped. In our simulations, senders buffer 1000 samples and transmit them in a single packet every 125 ms. Each outgoing packet contains a sequence number identifying the timing interval to which the packet corresponds. The receiver and intermediate merge points in the network expect to receive a packet from every upstream neighbor about every 125ms. 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.

We installed the audio mixing system as merge function on a network of Linux boxes that support the concast service. We used the topology shown in Figure 4 to emulate a system with a shared bottleneck link. The audio merge function was placed on the concast routers G_1 and G_2. We created simultaneous audio flows by starting sender applications (processes) at S_1 and S_2, so that two streams were sent from each toward the receiving application at R_1 (a total of four 64 Kbps streams). The concast routers G_1 and G_2 were each initialized with the audio mixer *merge*() function and a *getTag*() function that examined the sequence number. G_1 merges flows

[4]Note: the results shown in Section 5 were taken from our first implementation, which performs the (G, R, A) demultiplexing in the concast daemon rather than in-kernel.

from S_1 and S_2 as described above, while G_2 just passes the merged flows through unchanged. For our tests, a unique A value was sufficient to identify each flow and all FSBs were loaded with the audio mixer function described above.

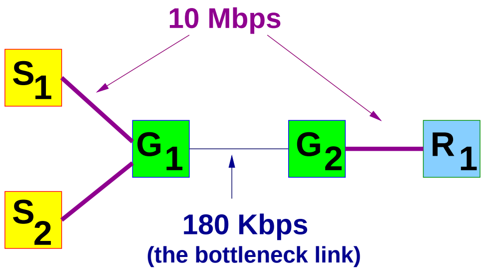

Figure 4: Network topology used in the audio mixing experiments.

To compare a concast-based implementation with the conventional unicast based implementation, we also implemented the audio mixer application using unicast messages sent to the receiver application. In this case, merging only occurred at the receiver.

Figure 5 shows the average number of senders included in the final audio stream delivered to the receiving application over each 1-second interval of the experiment. Results are shown for both the concast and unicast implementation. As expected, merging at G_1 ensured that no flow consumed more than 128 Kbps was sent over the bottleneck link, and thus concast was able to support all four simultaneous flows without losing audio from any of the senders. In contrast, the unicast implementation imposed a load of 256 Kbps on the bottleneck link, resulting in high loss rates. The erratic nature of the unicast delivery made it very difficult to maintain synchronization; occasionally multiple packets from the same source were included in the merged signal. This is the explanation for the spike above 4 at around 55 seconds in the unicast curve. In each experiment, we had two of the senders stop transmitting before the other two; thus both curves drop from 4 to 2 senders toward the end.

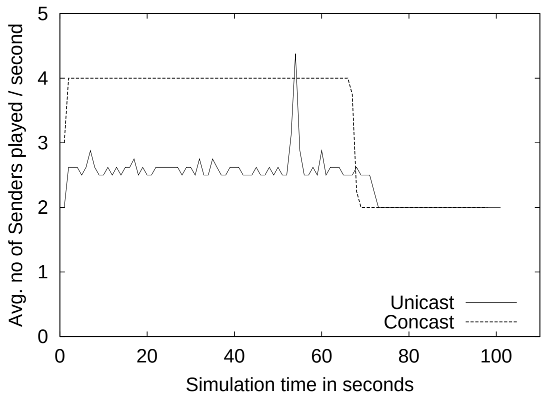

Figure 5: One-second moving average of the number of senders' signals played out at R_1 in each 125 ms interval.

6 Conclusions

Services that can be customized through a programming interface are likely to play a key role in future networks, although the forms of programmability that will be important are yet to be determined. We have defined a programmable network service called concast, which provides scalable many-to-one communication and allows applications to customize its semantics for merging messages. Our previous work demonstrated the benefit of concast through simulation [5]; currently we are implementing concast in the Linux kernel, and we expect to make that implementation available in the near future.

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, S. McCanne, and H. Zhang. An Application-Level Video Gateway. In *Proceedings of the ACM Multimedia Conference*, Nov 1995.

[4] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, August 1991.

[5] K. Calvert, J. Griffioen, A. Sehgal, and S. Wen. Concast: Design and implementation of a new network service. In *Proceedings of 1999 International Conference on Network Protocols, Toronto, Ontario*, November 1999.

[6] David R. Cheriton and Stephen E. Deering. Host Groups: A Multicast Extension for Datagram Internetworks. In *Proceedings of the 9th Data Communications Symposium*, pages 172–179. ACM/IEEE, September 1985.

[7] David. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[8] Michael Hicks, Pankaj Kakkar, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks, 1998.

[9] D. Katz. IP Router Alert Option, February 1997. RFC 2113.

[10] S. Kent and R. Atkinson. Security architecture for the internet protocol, November 1998. Internet Request for Comments 2401.

[11] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart Packets for Active Networks. In *1999 IEEE Second Conference on Open Architectures and Network Programming*, pages 90–97, March 1999.

[12] D.A. Wagner and S.M. Bellovin. A Bump in the Stack Encryptor for MS-DOS Systems. In *The Proceedins of the 1996 Symposium on Network and Distributed Systems Security (SNDSS'96)*, 1996. http://bilbo.isu.edu/sndss/sndss96.html.

[13] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.