Active Reliable Multicast on CANEs: A Case Study

M. Sanders[†], M. Keaton[‡], S. Bhattacharjee[♯], K. Calvert[♭], S. Zabele[‡], E. Zegura[†]

[†]Georgia Tech, [‡]TASC Inc., [♯]Univ. of Maryland, [♭]Univ. of Kentucky

Abstract— **This paper presents a case study in implementing a moderately complex, useful service on an active network platform. The active application is reliable multicast with congestion control; the platform comprises the Bowman Node Operating System and the Composable Active Network Elements (CANEs) Execution Environment. The importance of the work stems from the lessons it provides about the design and implementation of active platforms in general and Bowman/CANEs in particular. For example, our experience shows that timer-driven active node processing is as important as packet-arrival-driven processing. Thus, execution environments cannot focus exclusively on forwarding, but must also provide efficient timers and allow timer handlers the same capabilities as packet-driven computations. Other areas in which the implementation provides insight include service decomposition approaches for active applications and information sharing among service components.**

I. Introduction

Active networks provide a platform for network services that can be built or customized by injecting code or other information into the nodes of the network. For the purposes of this paper, the salient characteristic of "active networking" is the placement of user-controllable (for some definition of "user") computing capabilities in shared infrastructure of the communication network, where they can be utilized by applications that need those capabilities. This paradigm offers a number of potential advantages, including the ability to develop and deploy new network protocols and services quickly, and the ability to customize services to meet the needs of different classes of users.

As an example of a desirable network service, consider the problem of multicast applications that need reliability and desire to share bandwidth cooperatively with other applications. The present IP multicast service makes it difficult for applications to recover from losses in the network, because the needed information (location of losses) is hidden from users. Additionally, congestion control in the multicast environment is challenging due to the feedback implosion problem if all users communicate with the sender independent of one another. A number of active multicast projects have realized the value of placing the functionality where it is needed in the network—at the multicast branch points. The Active Error Recovery/Nominee-based Congestion Avoidance (AER/NCA) protocols realize multicast reliability and congestion control, taking advantage of active networking where it is present. The AER/NCA protocol design is based on current trends in reliable multicast research, but allows for enhanced performance when using active networks. Briefly, AER/NCA provides for rapid recovery from packet losses using packets cached at *repair servers* in the network, suppression of NACKs to avoid feedback implosion, and identification of the most congested receiver for congestion control.

AER/NCA was initially implemented in the ANTS Active Node Transfer System [3], arguably the most widely used environment for development of active network applications and services. While ANTS offers a number of advantages for code portability and rapid prototyping, it is not designed to support high performance, nor does it offer much explicit support for structuring complex protocols, beyond the support generally offered by a structured programming language. The AER/NCA service is a reasonably complex service; a significant portion of the processing occurs in the data path (e.g., retransmission of cached data packets), thus the implementation must provide reasonable performance.

This case study reports on the implementation of AER/NCA in an active network platform that includes the Bowman Node Operating System (NodeOS) and the CANEs Execution Environment (EE). Bowman is specifically designed to allow efficient access to low-level system resources, while CANEs is designed to provide a framework for structuring complex services. The overall platform attempts to strike a balance between performance and

This work was supported by DARPA under the CANEs project contract N66001-97-C-8512 and the PANAMA project contract N66001-97-C-8513.

flexibility.

The importance of this work stems from the lessons it provides about the design and implementation of active platforms in general and Bowman/CANEs in particular. For example:

- Our experience shows that timer-driven active node processing is as important as packet-arrival-driven processing. Thus, execution environments cannot focus exclusively on forwarding, but must also provide efficient timers and allow timer handlers the same capabilities as packet-driven computations.
- In the area of structuring complex protocols, we found that the CANEs composition mechanism provides a reasonably natural way to structure the AER/NCA service. Indeed, the service developers were able to decompose the service using standard modular programming and protocol techniques to map the components into the CANEs structure, comprised of a generalized forwarding function (GFF) and AER/NCA specific programs. This lesson may have bearing on the development of other EEs that fit the same general class (specifically, those based on customization of an underlying framework [8], [1]).
- We found a significant need for efficient packet duplication with copy-on-write semantics. Because this copy optimization is most often used on packet headers it should be possible to design a duplication method allowing the packet data to be parsed into pieces aligned with protocol headers, which are then cloned or physically copied as appropriate.

The next two sections provide an overview of the Bowman/CANEs platform (Section II) and an overview of the AER/NCA protocols (Section III), with an emphasis on providing just enough detail for the reader to understand the implementation experience[1]. The centerpieces of the paper are in Section IV, which describes design considerations and Section V, which discusses lessons learned in implementing AER and NCA on CANEs. In Section VI, we place our work in the context of related efforts. Section VII concludes the paper.

II. The Bowman/CANEs Platform

In this section we describe the aspects of the Bowman NodeOS and CANEs EE that are relevant for understanding the implementation of the AER/NCA protocols. Bowman and CANEs have been implemented at Georgia Tech and the University of Kentucky under the CANEs DARPA contract. More detail on Bowman can be found in [18]. More detail on CANEs can be found in [5].

[1]Perhaps as a testament to the complexity of the task, a fair amount of background is necessary to appreciate the implementation issues.

A. Bowman

Bowman was written out of necessity, to provide a platform for CANEs. At the time the Bowman development was beginning (Fall 1998) there were no NodeOS implementations available; indeed, the NodeOS specification [10] was still in considerable flux. For these reasons, Bowman does not implement the current NodeOS specification precisely, though it includes elements that are similar to those in the specification.

There are four elements of Bowman that are of interest; namely, *channels*, *a-flows*, *timers* and the *state store*. **Channels** in Bowman are communication endpoints that support sending and receiving packets via an extensible set of protocols. Bowman exports a set of functions enabling EEs to create, destroy, query and communicate over channels that implement traditional protocols (e.g., TCP, UDP, IP) in various configurations. In addition, Bowman channels allow other forms of processing such as compression and forward error correction to be included.

A-flows are the primary abstraction for computation in Bowman. Each a-flow consists of at least one thread and executes on behalf of an identified principal. The threads in an a-flow share context, in the form of variables that all threads can access. Bowman provides interfaces to create, destroy and run a-flows. As will be described later, one of the interesting questions in the implementation of the AER/NCA protocols is the division of functionality into multiple a-flows.

Associated with each a-flow is a dedicated input queue. A-flows request to receive packets that arrive on a particular channel by *subscribing* to the channel. Subscriptions contain a packet filter rule which allows Bowman's packet classifier to match the packets which should be placed in the a-flow's input queue. Figure 1 illustrates this process.

Bowman provides a **system timer** thread that a-flows can use to schedule processing to occur in the future. The interface to the timer facility allows set-timer and cancel-timer operations. The AER/NCA protocol makes extensive use of timers, for example to do negative acknowledgment (NACK) suppression.

The Bowman **state-store** provides a mechanism for a-flows to store and retrieve state indexed by a unique key. The Bowman state-store interface pro-

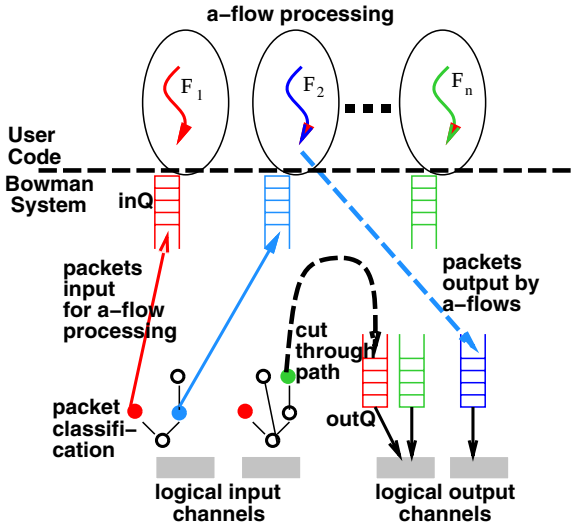

Fig. 1. Bowman channel subscription

vides functions for creating, storing and retrieving data from named state-stores. Using the underlying state store mechanism, Bowman provides an interface for creating named *registries*; such registries provide a mechanism for data sharing between a-flows without sharing program variables.

B. CANEs

The CANEs EE has two goals: to support the development of active applications which require reasonable forwarding performance, and to provide a framework for the modular construction of services. To meet these goals, CANEs allows programmability via customization of a basic packet processing loop. The programmer is constrained in that customizing code can only be added to certain locations in the basic loop; these constraints both encourage modularity and potentially allow for various hardware optimizations of the basic processing.

More specifically, the programming model used in the CANEs EE comprises two parts: a fixed part (*the underlying program*) that represents the uniform processing applied on every packet, and a variable part (*the injected program*) that represents customized functionality on the packets. The specific points in the underlying program where an injected program may be executed are called *slots*. Composition of services is achieved by selecting an available underlying program (e.g., generic forwarding) and then specifying a set of injected programs for customization. Injected programs may be node-resident or loaded from a remote site.

```
canes_recv(&pk);

c_Ep(cur_pkt) = pk;
c_Ep(route_this_packet) = TRUE;
c_Ep(do_post_processing) = TRUE;
// SLOT for Initial Processing
canes_raise_slot (PreProcess);
do_post_p = c_Ep(do_post_processing);
route_pk = c_Ep(route_this_packet);
pk = c_Ep(cur_pkt);

if (route_pk) {
 c_Ep(output_if_list) = NULL;
 // SLOT for Determining Output Channels
 canes_raise_slot(LookupRoute);
 ch_list = c_Ep(output_if_list);
 route_pk = c_Ep(route_this_packet);

 if (ch_list && route_pk)
  // forward on each channel independently.
  for (i = 0; i < ch_list->num_ch; ++i) {
   // copy only if the original is needed
   if (!do_post_p &&
    (i == (ch_list->num_ch - 1))) {
    pk_copy = pk;
    pk = NULL;
   } else
    pk_copy = o_pkt_copy(pk);

   c_Ep(cur_pkt) = pk_copy;
   c_Ep(output_if) = ch_list->ch_id[i];
   c_Ep(route_this_packet) = TRUE;
   // SLOT for Channel Specific Processing
   canes_raise_slot(Dispatch);
   route_pk_copy = c_Ep(route_this_packet);
   chid = c_Ep(output_if);

   if (route_pk_copy)
    canes_send(chid, &pk_copy, pk_copy->size);
   else
    o_pkt_free(pk_copy);
 }
}

if (do_post_p)
 // SLOT for Final Processing
 canes_raise_slot(PostProcess);

if (pk) o_pkt_free(pk);
```

Fig. 2. Example CANEs underlying program (GFF).

For example, Figure 2 shows an underlying program that can forward packets to one or more output channels. The program contains four slots: a PreProcess slot, a LookupRoute slot, a Dispatch slot, and a PostProcess slot. The PreProcess slot is raised for every incoming packet; the LookupRoute slot is raised once if the packet is to be routed, and allows customization of the routing function; the Dispatch slot is raised once per output channel, and thus allows per-output channel processing (e.g., to obtain the address associated with the interface); the PostProcess slot is raised after all routing is completed (e.g., to store a copy of the packet in a cache). We use the term *slot context tree* to refer to a computation consisting of an underlying program (the root) and set of injected programs. Because injected programs can raise additional slots, the tree may have arbitrary depth. When an injected program terminates, context is returned to the parent (program) in the tree and execution resumes at the point following the raise call.

An important issue regarding the CANEs EE is the design of underlying programs that have broad use. As conceived, we believed that CANEs might offer a small number of underlying programs. Our experience with AER/NCA supports this design decision: the same underlying program can be used for all of the parts of the implementation (see Section IV). However, additional experience with a variety of active applications is required before a firm conclusion can be reached.

The CANEs EE provides a variable sharing mechanism that is used by injected programs to communicate with underlying programs. Underlying programs declare shared variables, allocate space, and *export* variables to make them eligible for sharing. Injected programs declare *references* to shared variables. Further, injected programs *import* shared variables exported by underlying programs to create the binding between the shared variables and the references.

Figure 2 illustrates the use of shared variables from the underlying program point of view. For example, the variables `cur_pkt`, `route_this_packet` and `do_post_processing` are all shared. The underlying program exports and sets them using the `c_Ep()` operation prior to the raise slot function call. Any modification to the variables is recovered by copying into local, underlying program variables after the slot processing returns.

The instantiation of an underlying program and a set of injected programs at an active node occurs via the CANEs signaling protocol. The signaling protocol sends a CANEs user interface (CUI) message to the nodes which are to run the active application. A CANEs signaling a-flow is resident at any node supporting CANEs; when a CUI message is received, the underlying and injected program code is fetched and an a-flow is created.

III. The AER/NCA Protocol

A. Active Support for Reliable Multicast

The AER/NCA protocols are a result of the PANAMA DARPA contract. AER/NCA is implemented at the multicast sender and receivers, and also takes advantage of active network elements (active nodes) by placing *repair servers* at strategic points in the multicast distribution tree. A repair server handles both multicast packets from the sender and unicast packets sent to it by its downstream neighbors in the multicast tree, which may be receivers or other repair servers. A repair server also has the ability to "subcast" packets downstream, i.e. send a multicast packet to a limited portion of the multicast tree that is downstream of it. Repair servers are not required for correct operation of the protocols, but they improve scalability by distributing the processing load throughout the network and performance by responding to losses sooner.

The AER part of the service deals with reliability. It uses negative acknowledgments (NACKs) rather than acknowledgments, to reduce the amount of feedback from receivers to the sender. As the data packets flow down the multicast tree, they are intercepted by the repair servers. These packets are cached and used for repairing lost data packets as needed without involving the sender. Losses are detected by both receivers and repair servers, to allow faster recovery. Each participant has an *upstream neighbor*, which is either the sender or a repair server, whichever is closer. When loss is detected, NACKs are unicast to the upstream neighbor. NACK suppression is performed using well-known random back-off and subcast techniques [11], [21] to reduce the amount of NACK traffic. Repair servers propagate NACKs upstream as needed. Retransmissions from either the sender or a repair server are multicast or subcast downstream, with downstream repair servers filtering retransmissions so that only those parts of the tree that need the information will receive it.

The NCA component of the service implements congestion control with a sender-based rate adjustment algorithm, using packet loss indications from a single

nominee receiver to regulate the transmission rate. It is designed to be TCP-fair, which informally means that the AER/NCA session must not receive more bandwidth than competing TCP sessions on any of the links in the multicast tree. NCA has two more or less independent subcomponents: nominee selection and rate control. Only the former involves the active nodes; rate control operates between the nominee receiver and the sender.

The nominee selection process periodically selects a "worst" receiver (i.e. a receiver to which the path from the source is most congested) by means of congestion reports sent upstream. Repair servers aggregate these congestion reports and forward only the "worst" one. The sender receives a small number of congestion reports and selects the nominee from among them. Once a nominee receiver has been selected and activated, the NCA rate control protocol can begin regulating the sender transmission rate.

B. Component Protocols

AER/NCA is implemented as a collection of algorithms and protocols working in concert to support reliable multicast. In this section, we present details of some of the component functions, focusing on those that involve processing (other than forwarding) at active nodes. A complete description of the protocol is available at [4]. The AER/NCA protocols are designed to perform the following general functions:

- Establish and maintain the tree of AER nodes (hierarchy of repair servers) on the multicast path from source to the receivers, and enable each node to learn the identity of its parent repair server.
- Support reliability by caching out-bound data packets and retransmitting them when NACK messages are received. Also, subcast NACKs back to the portion of the multicast group from which they originate, and other processing to aid in NACK suppression.
- Estimate various round-trip times for use in the different algorithms. (E.g. before transmitting a NACK for a lost message, receivers delay a random amount of time that is a function of round-trip-time.)
- Identify a "nominee" receiver, i.e. the one experiencing the greatest loss rate.

In the following subsections, we give an overview of each of these functions, with an eye toward mapping the functions into the CANEs model.

B.1 State Establishment and Maintenance

The multicast source periodically sends a source path message (SPM), which causes soft state to be established for the AER session at each hop along the path from source to receivers. The SPM message originally carries the source's address; each active node replaces the carried address with its own before forwarding, and uses the old carried address as the identity of its *repair server*, as illustrated in Figure 3. (Note that for some active nodes the repair server will be the source.)

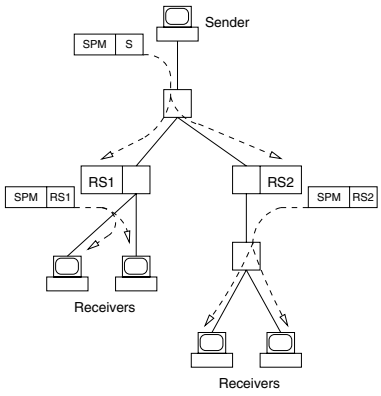

Fig. 3. AER SPM operation.

Several parts of the loss recovery mechanism (see below) rely on the ability of repair servers to *subcast* on the forward path, i.e. to send messages that go to the portion of the multicast group downstream from that node. The state information established by this part of the protocol makes that possible. The state associated with the session times out and processing ceases if no SPM is received for some time; this enables the AER tree to adjust to topology changes.

B.2 Reliability Mechanism

The AER reliability mechanism is responsible for detecting lost packets and supplying *repair packets* for them in a low-latency, efficient manner. Both receivers and repair servers are capable of detecting lost data packets; the sender and repair servers are all capable of supplying repair packets. Data packets contain sequence numbers; losses are detected via gaps in the received sequence. As each data packet is forwarded by an active node (repair server), a copy is made and placed in a cache. Repair servers also monitor the sequence number progression in order to detect losses early.

Upon detecting a lost packet, a receiver or repair server waits a random interval for NACK suppression purposes. If it has not received a NACK for the same lost packet by the end of the back-off interval, it sends a NACK containing the sequence number of the missing packet upstream to its repair server. If it does

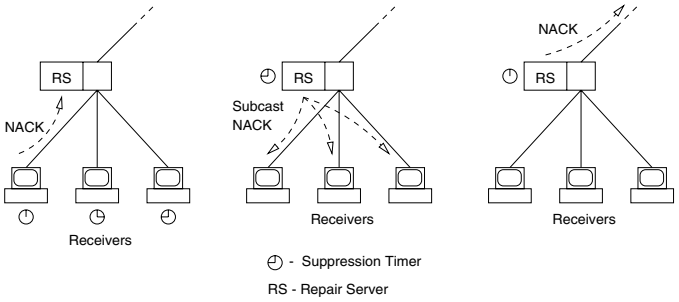

Fig. 4. AER NACK processing.

receive a NACK for the same packet during the back-off interval, it suppresses its transmission but proceeds as if it had sent the NACK. A repair server, upon receiving a NACK, performs one of two things. If the repair server has the needed repair packet in its cache, it immediately subcasts it to the receivers, which completes the repair procedure. However, if the repair server does not have the needed repair packet in its cache, it subcasts the NACK back to the receivers for which it is the repair server to suppress further NACK packets for the same repair packet. After a random suppression delay at the repair server, a NACK is unicast upstream to *its* repair server. This process is displayed in Figure 4.

Repair data packets and subcast NACKs only travel down one level in the multicast tree. Repair servers that receive a NACK from upstream process the packet (to suppress their own NACKs) but do not forward it. Repair servers that receive a subcast repair packet cache it and subcast it further only if a NACK for that packet is pending. Losses in the repair process —i.e. lost repair packets or lost NACK packets— are detected by means of timeouts based on RTT estimates. If a NACK was sent and the requested repair packet fails to arrive in the specified time, the NACK is retransmitted to the node's repair server (again using the randomization procedure to suppress duplicates). The repair server distinguishes duplicate NACKs from retransmitted NACKs by means of a counter included in each NACK packet. Each time a NACK for a particular data packet is retransmitted, the receiver or repair server sending the NACK increments the counter. Repair servers keep track of the highest counter value seen for each sequence number, and ignore NACKs that contain smaller values.

The use of suppression and aggregation techniques, and the use of a hierarchy of repair servers ensures that (i) sender and repair servers are not subject to implosion; (ii) transmission of unnecessary repair data packets is minimized; and (iii) repair latency is kept small.

B.3 Round-Trip-Time Discovery

Detection of lost NACKs and repair packets requires that each node have an estimate of the round-trip-time (RTT) between itself and its repair server. In addition, two other RTT values are needed at each entity: the "maximum peer group RTT", which is the maximum RTT over all nodes using the same repair server, and the sender RTT, which is the round-trip time between the node and the sender.

These estimates are computed by having each node periodically send a Get-RTT request packet upstream to its repair server, containing a locally-generated time-stamp. Repair servers reply with a Get-RTT response packet containing the original time-stamp and sent to the source of the request packet via unicast. The round-trip-time for this message is then used in the RTT estimate.

In addition, each Get-RTT request contains the originator's *current* RTT estimate; the repair server computes and stores the maximum estimate from all Get-RTT requests, and returns the current value in every Get-RTT response. The Get-RTT response also carries an estimate of the repair server's RTT to the sender; the sender sets this to zero. When it receives the Get-RTT response, a node adds the repair server's to-sender RTT to its own (new) to-repair server RTT, to obtain its own to-Sender estimate.

B.4 Nominee Identification

The NCA protocol implements congestion control for the multicast session. The basic idea is that the rate of transmission is governed by the worst loss rate among receivers; that loss rate causes the sending rate to be modulated in a manner similar to that of TCP-NewReno [13]. Thus there are two algorithms: one for selection of the nominee, and one for modulation of the transmission rate. The active nodes participate only in the selection of the nominee; hence we only describe that algorithm.

NCA attempts to determine the TCP-fair bandwidth on the path to the most congested receiver. NCA uses the TCP throughput equation [20] to determine the fair rate; the use of the equation requires the loss probability and the round-trip-time to the most congested receiver. The protocol operates as follows. Each receiver maintains an estimate of its loss probability estimate, p, as well as its round-trip-time to the sender, T. These values are periodically reported

to its repair server, which aggregates received (p, T) values and forwards only the worst receiver's parameters upstream to its repair server. This aggregation continues up to the sender, where the final decision is made as to which receiver is the nominee. The sender then uses normal unicast methods of notifying the old and new nominees.

IV. Implementation Experience

In this section we primarily focus on the design considerations that arose in mapping AER and NCA onto the CANEs platform. We finish the section with a description of our test topology.

A. Mapping AER/NCA onto CANEs

AER/NCA accomplishes the four general functions outlined in the previous section using eight different packet types. This level of complexity and protocol interaction represents exactly the kind of "composite" service for which CANEs was intended. It is therefore crucial to consider the possibilities for structuring the code bindings to the slots in one or more underlying programs, as well as deciding what slots to raise within injected programs. As with any structured design, there is a tradeoff between performance and the potential for code re-use via modularity.

Recall that in CANEs, the basic unit of functionality at an active node is the a-flow; an a-flow is specified by an underlying program, a set of injected programs, and the bindings of injected programs to slots in the underlying program or other injected programs. We first consider the a-flow structure.

One possibility is to implement the entire service as a single a-flow, which examines each packet via a *PreProcess* slot to determine its type, and then simply invokes the appropriate function for that type. The primary advantage of this method is a reduction in the overhead of crossing "module" (in this case a-flow) boundaries when interaction between the different functions is required. The primary disadvantage is the classical one of reduced flexibility: no component can be changed without rebuilding the entire package, and individual pieces cannot be re-used. For example, the tree-establishment and maintenance portion of the protocol might be useful for media-thinning or other applications that need to perform processing hop-by-hop.

The approach we used was to consider the processing of each distinct packet type as a "micro-protocol" [19], with one a-flow per packet type. This choice was to some extent motivated by restrictions on the capabilities of individual threads within an a-flow in Bowman. However, it does leverage the demultiplexing capability of the NodeOS, and makes re-use of any component protocol straightforward.

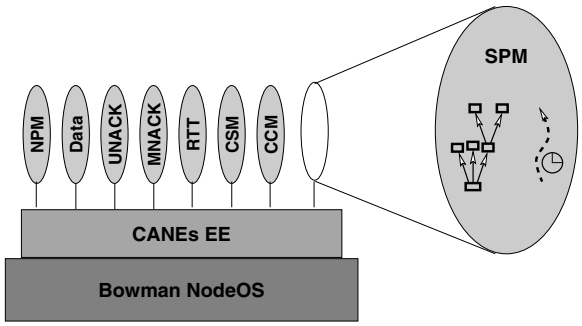

Fig. 5. AER/NCA a-flows.

Figure 5 shows the a-flows associated with the AER/NCA protocol; each is labeled with the packet type it handles. The blow-up of the SPM a-flow reveals the existence of a dedicated underlying program thread and a timer thread; each a-flow has this same structure. Table I gives a brief description of the function of each a-flow, including whether it processes unicast or multicast messages going upstream, downstream or in both directions.

a-flow	Function	Direction
SPM	source path maintenance	down/multi
Data	data forward/cache	down/multi
UNACK	NACK transmission	up/uni
MNACK	NACK suppression	down/multi
RTT	round-trip-time	both/uni
CSM	nominee selection	up/uni
CCM	nominee feedback	up/uni
NPM	nominee path (not used)	up/uni

TABLE I
AER/NCA packet types.

Figure 6 shows the decomposition of the SPM and Data micro-protocols and the resulting context trees of bindings between raised slots and injected programs. The root of each tree is the underlying forwarding function. As shown in Figure 2, the GFF has four slots that can be raised. In the case of the SPM a-flow, three of the slots are used; the Data a-flow uses two. Notice that both the `spm_ppp` and the `data_ppp` injected programs also have slots, to which the routines that handle all of AER's interaction with a packet cache (e.g., *PktCacheGet*) are bound. This allows different instances of AER to use

different caching mechanisms and/or policies while running on the same node.

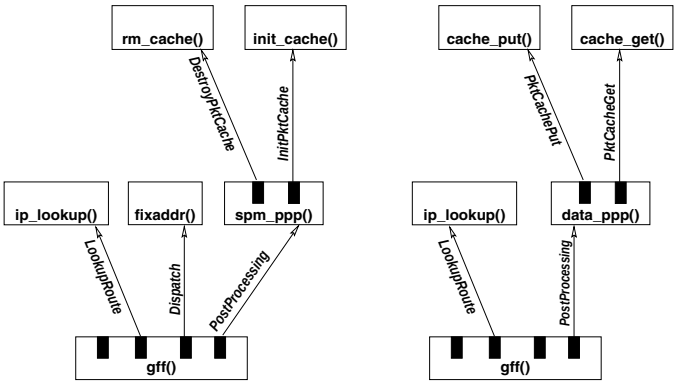

Fig. 6. SPM(left) & DATA(right) slot context trees.

Both the SPM and Data a-flows are arranged so that most of the non-forwarding work occurs in the *PostProcessing* slot. This allows packets to be forwarded first, so that active processing occurs concurrently with queuing, propagation, and other outbound delays. Of course, if the outgoing packet differs from the incoming packet, it is not possible to do *all* processing after forwarding. For example, the SPM micro-protocol has to replace the "previous hop" value in the received packet with its own address before forwarding the packet, because it defines its downstream neighbors' next hop on the upstream path toward the source. This replacement is done in the *Dispatch* slot, which the GFF raises once for each outgoing interface. The `fixaddr()` injected program is thus able to place the address of the particular outgoing interface in the packet, as it should.

Table II shows the total number of different injected programs bound to each of the four slots in the GFF shown in Figure 2. The *Other* row reveals the bindings made to slots raised by the AER/NCA specific injected programs. One a-flow, the CCM packet processing, uses just one of the slots; all others use two or more slots.

Slot	Number I.P.'s Bound
PreProcess	5
LookupRoute	8
Dispatch	2
PostProcess	2
Other (User-defined)	5

TABLE II
AER/NCA USE OF PROGRAM SLOTS.

B. Implementation evaluation setup

Following the initial protocol development effort, we developed a topology and test scenarios for evaluating and improving our work. Our primary goal was to design a setup capable of performance and stability testing of the protocol and the platform. Secondarily we desired a topology which allowed confirmation and demonstration of the advantages of the AER/NCA protocol.

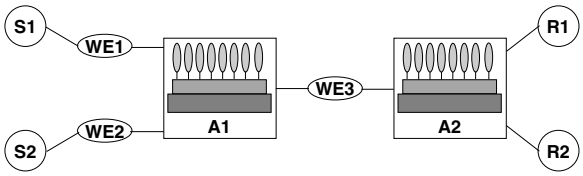

Fig. 7. Experimental Topology.

The topology in Figure 7 contains NISTNet [2] WAN-emulators (WE1, WE2, and WE3) which allow delays, drops, and bandwidth limitations to be introduced on a link. This capability is necessary if the active protocols are to be useful. The topology contains two MPEG-2 video sources S1 and S2 attached to active router A1 via WAN-emulators WE1 and WE2 respectively. Two receivers R1 and R2 are attached to active router A2. The active routers are connected to one another via WE3. The receivers contain streaming MPEG-2 hardware decoders; the active routers have dual processors. The video sources transmit MPEG-2 video in the half-D1 (360 x 480) format at a full 30 frames per second. The data rate for a single video flow averages 2 Mbps with frequent bursts of up to 6 Mbps.

The active routers are able to easily route multiple multicast video streams from either source simultaneously to both clients while running the AER/NCA active applications. We are able to demonstrate an obviously noticeable video quality difference between an active receiver that gets repairs from A1 or A2, and a non-active one that only communicates with the source. We also have a test server and receiver application that uses the AER/NCA protocol to communicate. This test application allows rate and timing adjustments to be made for more thorough exercising of the code and the protocol. It is important to note that this isolated topology not only provides the ability to run reproducible experiments at rates sufficient for streaming video; but also provides fine-tuning of link properties which is necessary for localization and debugging the limiting factors of an implementation.

V. Lessons Learned

In this section we highlight the lessons learned in the implementation and propose additional CANEs features and modifications. Some of these proposed changes have been made; others require additional thought or implementation effort.

A. Timer-driven processing

Prior to the implementation experience, we had not fully appreciated the importance of timer-driven processing. We suspect others in the active network community have similarly focused on packet arrivals as the primary event of interest. The AER/NCA protocols require that timer events be processed efficiently and have similar capabilities to those of packet arrival events. For example, in the NACK suppression and retransmission cases require randomly delayed actions as well as the ability to generate and forward packets. While packet arrival does trigger these actions, a timer facility is necessary if NACK implosion is to be avoided. AER/NCA makes significant use of the timer facility provided, two per-flow timers are used along with per-message or per- sequence number timers for NACK, CSM, and RTT processing.

The original CANEs timer facility was inefficient because there was a single system thread that executed all timer handlers. This meant that timer handlers could not block or execute for long periods of time without delaying other handlers. Because much of AER/NCA's functionality is timer-driven, we were led to develop a "heavyweight timer" facility, which is executed in a separate thread associated with the a-flow which set the timer. The system timer thread signals the heavyweight timer thread, which executes the handler.

Perhaps more subtle, and somewhat more specific to CANEs, we found that the newly created heavyweight timer facility is of limited use without the ability to raise slots within timer handlers. For example, NACK packets can be sent upstream as a result of either packet arrival or timeout. Ideally we would like to raise a slot to lookup and forward the timer-driven NACK packets in the timer handler; however, that is not possible because the handler executes on the timer thread, which has no slot context. Rather than specifying a function name as a timer handler, the solution to this problem is to allow a timer handler to be specified with the *name of a context tree*, where the root of the context tree is executed as the timer handler.

Providing the ability to execute named context trees in timer handlers requires support that we have not fully designed and implemented, including deciding on a name space. There are also implications for code loading, and signaling, as these affect when and how the code for the timer handler is loaded into the appropriate nodes.

B. CANEs use of a-flows

We intended the a-flow to be a conceptual grouping of related threads for the purposes of security, accounting, and scheduling. The use of a separate a-flow for each sub-protocol, combined with the timer thread per-a-flow described above, violated the a-flow semantic by distributing macro-protocol functionality across multiple flows. In addition, thread density lead to significant contention for protocol state between concurrently running threads. We discuss each of these challenges in the two paragraphs below.

To achieve the desired conceptual grouping, future CANEs a-flows should allow the specification of an arbitrary collection of threads. Slot context should be specifiable for each thread, and packet arrival threads should each have an I/O specifier, potentially giving each thread it's own input queue. Allocation of zero or more of these threads for the purposes of timer handling should be supported. For example, the allocation of timer threads might be based on issues of concurrency and contention for shared variables. Thread pooling in the form of packet handlers or timer handlers should be supported as well.

The NACK state associated with an AER/NCA flow demonstrates the contention issue; this state is referenced by four of the micro-protocols. When we account for timer threads, as many as six threads can be competing for NACK state at the same time. These threads may also require access to other state information, for example the packet cache, which can lead to additional race conditions as well as state-store lookups. In the case of AER/NCA we were able to find effective state objects and lock scope by basing object granularity on both the reference frequency and the possibility of sub-protocol contention. Additionally, the overhead of multiple state-store lookups per event was avoided by storing related state references within all objects associated with the flow. For example, the resulting state object for NACK state is lockable and hash-able on a per sequence number basis and contains a pointer to the AER/NCA state object for the flow. This in turn allows access to all other state for the AER/NCA flow without the need for an

additional state-store operation.

C. Packet duplication

Platforms which support parallel execution of active applications or allow the caching of packets, will require a duplication semantic which minimizes the copy operation overhead while allowing flexibility in packet modification. In the case of CANEs, the injected programs that run in different a-flows need to share packet contents with one another as well as the system threads responsible for forwarding. For example, the SPM message rewrites the previous hop field for each outgoing interface. However, the CANEs duplication facility currently provides only monolithic "reference count" semantics, i.e., there is one actual copy of the data, which is shared by all programs. This makes duplication fast, but also means that any modification applies to all copies. Obviously this causes problems if an SPM message is still queued for output on one interface when the address is rewritten for another.

To get around this problem, the CANEs duplication method was modified to copy the entire packet. However, a better solution is a copy-on-write duplication method that allows sharing of copies at the granularity of individual headers, so that only the parts of the packet being modified are copied. Because this copy optimization is most often used on packet headers it should be possible to design a duplication method allowing the packet to be parsed into pieces aligned with protocol headers, which are then cloned or physically copied as appropriate. The portion of the packet of interest to the caller could be defined using the Bowman packet filter's header definition language. The kind of active processing discussed here is further evidence of the need for previously proposed buffer manipulation ideas such as fbufs[9].

D. Interoperability with non-active end-points

Active network platforms will need to facilitate communication with non-active end-points in order to support incremental deployment and transparent inclusion of active network functionality. In implementing the AER/NCA protocols we were ultimately interested in demonstrating the service using a video server and receivers. Due to the complexity of the video components, it was considered a necessity to use an existing video application without modification. Packets sent from the end-points needed to traverse the CANEs/Bowman active network with their UDP and IP headers intact so that they could be demultiplexed correctly by the receiving application.

In CANEs/Bowman, application-level demultiplexing of packets to a-flows is separated from channel processing. That is, the application specifies the packet filter for the packets that emerge from the Bowman input channel, independent of the type of that channel. This means that the same filter expression can be used with all channels regardless of their type.

Because the active nodes' point of view is that packets from the end-points are transmitted directly over Ethernet, we solve the problem with channels defined by the Ethernet MAC addresses of both the end-point and the interface on the active node. We also modify the routing tables on the end-points so that packets destined for all other end-points and active nodes in the topology are routed through the adjacent active node. As a result, no other knowledge of the active network is necessary on the end-points.

E. Flexible run-time control

In development and testing we encountered the need for a significant diversity of experimental setups. We were able to exert considerable run-time control over the setup via the CUI signaling interface. This not only saves time otherwise spent compiling, but also allows policy-oriented design decisions to be made at run-time rather than design time.

For example, one of the earliest design decisions we faced was whether to limit a particular AER/NCA protocol instantiation on a node to use by a single application flow. Because the flow classification section of the CUI message allows for either very specific (e.g., source and multicast addresses) or very general (e.g., all AER packets) filters to be specified, we opted not to restrict the application to a single flow. As a result, we built our active applications to work independent of this policy decision; a single instantiation of the protocol can either support multiple flows simultaneously or only one specific flow, by simple alteration of the signal at run-time.

The run-time specification of the slot bindings also allows flexibility as to which of the sub-protocols were activated, and which injected programs were used. In one case, we added a slot for a lightweight/low-frequency flow-authorization mechanism in the SPM packet handler to demonstrate security capabilities. The use of this mechanism could be completely avoided at run-time by simply not binding the injected program to the slot added to the SPM handler.

F. Evolving a generalized forwarding function

Our original CANEs GFF turned out to be oversimplified. We found that refining a generalized forwarding function was a difficult task due to the tradeoffs between efficiency and flexibility.

The original GFF contained only two slots, one for pre-processing and another for route lookup. AER/NCA required a number of changes, including support for multicast, the ability to process packets after they have been forwarded, and the ability to process channel-specific copies of packets being forwarded. This lead to the development of the GFF in Figure 2. Multicast support was interesting since we wanted to make the forwarding function general enough to handle multicast, while minimizing any unnecessary operations in the case of unicast.

VI. Related Work

Most closely related to this work are other efforts to develop moderately complex active applications and services. Of particular interest are those that involve programmability on the data path and have some performance requirement. Efforts in this area have been limited for two main reasons. First, there have been relatively few active network platforms available that emphasize performance. Second, quite a bit of active service and application effort has concentrated on adding programmability to the control plane, rather than the data plane.

We mention several projects that have produced (or are working on) moderately complex applications and services. This discussion is by no means exhaustive, but simply reflects the more substantial efforts that we are aware of.

The Router Plugins project at Washington University involves the development of an in-kernel execution environment that allows limited customization of IP forwarding [8]. The execution environment has been implemented in FreeBSD and offers a relatively high performance platform for data path programmability. The platform has been used to implement congestion control mechanisms for best-effort multicast video distribution [16]. These mechanisms work in concert with a wavelet encoding scheme to preserve as much quality as possible under congested conditions. This work demonstrates the potential for data path programmability to substantially improve end-user performance.

The Active ARP project at ISI is investigating the use of active networking to customize resource reservation, thus the focus is on programmability in the control plane. As part of this project, an EE is being developed specifically for this "application". This work certainly meets the test of significant complexity in the service; indeed, this is one of the most ambitious service development efforts currently underway in the active and programmable networks community. The lessons learned from the project will be of great interest.

The ANTS execution environment has provided the platform for quite a variety of application and service development efforts [23], including PIM-style multicast and Web cache routing [24]. At least one of the lessons from these implementation efforts is consistent with our findings, namely that the active processing tends to be quite efficient, with a very modest slow down over null active processing.

Although the main contribution of this paper is not reliable multicast, we briefly mention related work in this area. A number protocols have attempted to achieve multicast reliability and congestion control without additional support from the shared nodes of the network, that is, using only unicast and multicast communication among the end systems [11], [12]. However, they all have a number of inefficiencies when dealing with large numbers of receivers, due to the need for feedback from the receivers to the sender. As the group gets large this feedback can overwhelm the sender, a condition known as *implosion*. Second, as the number of receivers grows, the number of retransmissions grows [6], which can lead to congested links and degradation of receiver performance. Developing a congestion control algorithm allowing multicast traffic to share bandwidth with existing unicast applications over an entire multicast tree turns out to be a complex problem [22], [14], [7], [25], [15], [17]. Thus end-system-only approaches either don't scale or end up constructing an "overlay" topology, because the actual multicast tree topology used by the IP multicast service is hidden.

VII. Conclusions

In this paper we have described an implementation of the AER/NCA reliable multicast scheme over the CANEs/Bowman active network platform. During the implementation, we gained valuable insights about structuring non-trivial protocols over active platforms. Our effort demonstrates the feasibility of implementing *real* protocols over active networks.

This implementation effort has enhanced our understanding of both the CANEs/Bowman platform

and of AER/NCA. None of the earlier active services (e.g. a "smart" firewall) implemented in CANEs stressed the timer handling routines as rigorously as AER/NCA. The AER/NCA implementation exposed the inevitable bugs in our timer routines, but more importantly, forced us to re-design timer support in CANEs. As mentioned before, this led to the addition of a multi-threaded "heavy-weight" timer facility in CANEs. Of course, in retrospect, it is clear that any moderately complex protocol will require substantial timer support from the environment, but it is instructive to note that when the port of AER/NCA to CANEs was started many EEs did not provide *any* timer support. Our experience with the timers is also a perfect example of the application requirement–EE feature feedback loop[2] that had to be set up in order to implement AER/NCA over CANEs/Bowman.

AER/NCA consists of eight co-operating micro-protocols. Our implementation effort taught us valuable lessons about the features CANEs (and other EEs) must support in order to provide a natural environment for protocol (de-)composition. The CANEs notion of an underlying program and slots into which micro-protocols can be inserted as injected programs proved to be relatively natural and useful in decomposing AER/NCA. The current CANEs model (as described in [5]) is very good at decomposing protocols into different execution threads, but the structure would have been better had we not used an a-flow per micro-protocol. Instead, the real requirement is for CANEs to be able to support complete multi-threading with the ability to raise slots within each thread. In this way, individual threads can be used to decompose the protocol while the usual synchronization primitives, e.g. semaphores and condition variables, can still be used for synchrony.

References

[1] ASP EE in the ABone. http://www.isi.edu/abone/ASP_EE.html.

[2] Nistnet. http://snad.ncsl.nist.gov/itg/nistnet/index.html.

[3] Active Node Transfer System Version 1.3, April 2000. http://www.cs.washington.edu/research/networking/ants/.

[4] AER/NCA Use Cases, April 2000. http://www.tascnets.com/newtascnets/Software/AERNCA/.

[5] S. Bhattacharjee. *Active Networking: Architecture, Composition, and Applications*. PhD thesis, Georgia Institute of Technology, Aug. 1999.

[6] S. Bhattacharyya, J. Kurose, and D. Towsley. The Loss Path Multiplicity Problem in Multicast Congestion Control. In *Proceedings of IEEE INFOCOM*, March 1999.

[2]Our thanks go to Doug Maughan at DARPA for encouraging and facilitating this collaboration.

[7] D. Chiu. Congestion Control using Dynamic Rate and Window. Technical report, Presented at the Meeting of the Internet Reliable Multicast Research Group, Arlington, VA, December 1998.

[8] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of SIGCOMM '98*, Vancouver, CA, Sept 1998.

[9] P. Druschel and L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 189–202, December 1993.

[10] L. Peterson (Editor). NodeOS Interface Specification. DARPA AN NodeOS Working Group Draft, 1999.

[11] S. Floyd et al. A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. In *IEEE/ACM Trans. Net., vol. 5, no. 6*, pages 784–803, December 1997.

[12] S. Paul et al. RMTP: A Reliable Multicast Transport Protocol. In *IEEE JSAC vol. 15, no. 3*, pages 407–421, April 1997.

[13] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, experimental, April 1999.

[14] S. Golestani and K. Sabnani. Fundamental Observations on Multicast Congestion Control in the Internet. In *IEEE Infocom'99*, March 1999.

[15] M. Handley and S. Floyd. Strawman Specification for TCP Friendly (Reliable) Multicast Congestion Control (TFMCC). Technical report, Presented at the Meeting of the Internet Reliable Multicast Research Group, Arlington, VA, December 1998.

[16] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, and B. Plattner. An Active Router Architecture for Multicast Video Distribution. In *Proceedings of Infocom 2000, April 2000, Tel Aviv.*

[17] M. Luby, L. Vicisano, and T. Speakman. Heterogeneous Multicast Congestion Control based on Router Packet Filtering. RMT working group, June 1999.

[18] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert. Bowman: A Node OS for Active Networks. In *Proceedings of Infocom 2000, April 2000, Tel Aviv.*

[19] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[20] T. Ott, J. Kemperman, and M. Mathis. *The Stationary Behavior of Ideal TCP Congestion Avoidance*. August 1996.

[21] C. Papadopoulos, G. Parulkar, and G. Varghese. An Error Control Scheme for Large-Scale Multicast Applications. In *Proceedings of Infocom 1998.*

[22] L. Rizzo, L. Vicisano, and J. Crowcroft. TCP-like Congestion Control for Layered Multicast Data Transfer. In *IEEE Infocom'98*, April 1998.

[23] D. Wetherall. Active Network Vision and Reality: Lessons From A Capsule-Based System. In *17th ACM Symposium on Operating System Principles (SOSP'99)*, December 1999.

[24] D. Wetherall. *Service Introduction in an Active Network*. PhD thesis, MIT, February 1999.

[25] B. Whetten and J. Conlan. A Rate Based Congestion Control Scheme for Reliable Multicast. Technical report, Presented at the Meeting of the Internet Reliable Multicast Research Group, Arlington, VA, December 1998.