Designing Service-Specific Execution Environments

Mary Bond, James Griffioen, Chetan Singh Dhillon, and Kenneth L. Calvert*

Laboratory for Advanced Networking
University of Kentucky
Lexington, KY 40506

Abstract. Past work on the *active network architectural framework* has focused on the NodeOS and Execution Environment (EE), which offer rather standard programming environments. As a result, *Active Applications (AAs)* must build the desired service from the ground up. Ideally, active applications could be built on higher-level *active services* that could themselves be "programmed". In this paper, we examine the issues involved in the design and implementation of higher-level active network services. We describe the issues that arise when using AAs to implement these services and then present our experiences implementing one such service, namely *concast*, as an AA running on the ASP EE. The paper concludes with performance numbers obtained from an example audio-merge application that shows the viability of using AAs as specialized execution environments.

1   Introduction

Although the active network community has designed and largely embraced a common architectural framework for active networks [6], we are only now beginning to gain experience building active network applications that use the framework. This paper reports on our experiences building applications using the framework. In particular, we consider the issues involved with implementing higher-level services which we simply call *active services (AS)*. In some sense, active services are analogous to the upper layers of the IP protocol stack, offering more specialized services to applications than the basic network-level service (IP). We consider the requirements of active services, and discuss the suitability of the architectural framework for supporting these types of services.

The active network framework, illustrated in Figure 1, is quite general. It consists of a set of nodes (not all of which need be active) connected by a variety of network technologies. Each active node runs a *Node Operating System* and one or more *Execution Environments*. The *NodeOS* is responsible for allocating and scheduling the node's resources (link bandwidth, CPU cycles, and storage), while each *Execution Environment (EE)* implements a virtual machine that interprets active packets arriving at the node.

Each EE defines a different virtual machine or "programming interface" on which *Active Applications (AAs)* can be built to provide a particular end-to-end service. One EE may provide a completely general virtual machine (e.g., Turing complete) while another provides a more restrictive virtual machine. However, in both cases the programming environment provided by an EE is intended to be useful to a wide range of applications. AA code, on the other hand, is designed to meet the specific needs of a particular application.
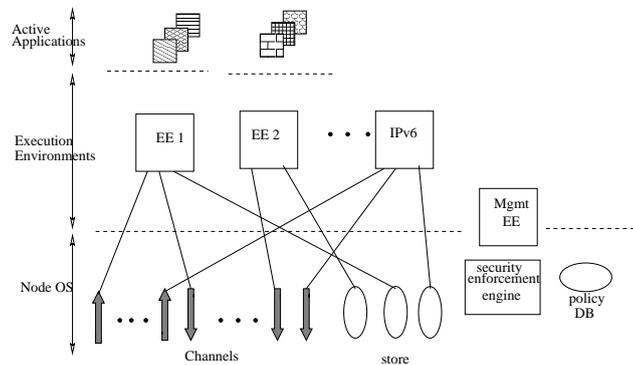

Fig. 1. The Architectural framework for an Active Network.

In the current Internet, applications are rarely built directly on the IP protocol. Instead, a *protocol stack* of higher-level protocols offer common/useful services (e.g., TCP, UDP, RPC, XDR, VPNs, RTP, etc.) needed by a wide range of applications. A similar need for high-level services arises in the context of active networks. These higher-level services may themselves be customizable, offering a special-purpose programming model and execution environment. For example, a "packet thinning" service (AS) might allow applications to define the function that selects packets to be dropped. Because packet thinning is a "higher-level" service, the packet thinning programming environment is much more restrictive (i.e., tailored to packet thinning) than the general programming environments offered EEs. Moreover, the active service typically performs the task of deploying and installing the user's function only at nodes that need it. Currently, the active network framework is agnostic about how such services should be implemented within the active network architecture.

One option is to implement each active service as an EE (see Figure 2(a)). However, given the specialized nature of an AS, it does not really fit the generic service model of an EE. Moreover, because EEs tend to be general-purpose programming models, development and deployment of an EE is considered a nontrivial, heavyweight exercise, and it is expected that the number of different EEs supported will be small. We expect there will be tens, possibly hundreds, of different active services which would imply a proliferation of EEs and clearly goes against the principles of the framework.

Another alternative is to implement each AS as an AA (Figure 2(b)). A clear advantage of this approach is that an AS does not need to re-implement resource allocation

mechanisms or policies, but rather can leverage these services from the underlying EE. However, unlike normal AAs, "AAs used as ASes" must also support a programming environment capable of running application-specific code (i.e., other AAs). Followed recursively, this approach leads to AAs acting as execution environments for other AAs who may, in turn, act as even more specialized execution environments for other AAs (i.e., similar to a protocol stack).

(a) An AS implemented as an EE.

(b) An AS implemented as an AA.
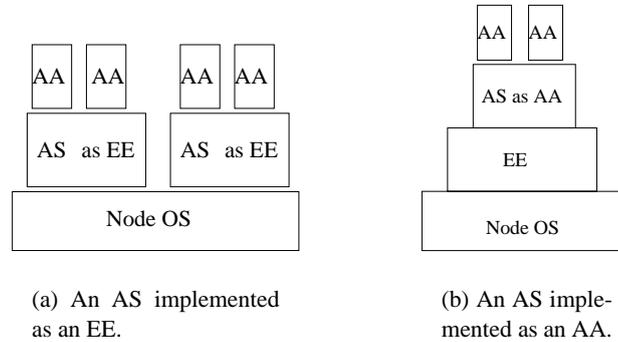
Fig. 2. AS implementation approaches.

A third approach which has not received much attention is to implement AS services as libraries linked against the AA. The resulting AA is then loaded into the EE and implements the AS functionality only for that application. This has some obvious drawbacks, including the need to load multiple copies of the same AS code, the AA being tied to an EE rather than an AS, and the difficulty of managing state and allocating resources across flows (AAs).

Given the drawbacks associated with implementing ASes as EEs or as a library, this paper explores the advantages and disadvantages of implementing ASs as AAs. We describe an implementation of a *Concast AS* executing as an AA on the ASP EE [1]. Although we show experimental results from an audio application that uses the concast AS, the main contributions of the paper is the experience gained implementing an AS as an AA. In particular, we found that many of the EE services had to be replicated by the AS. In some cases, re-implementing these services as an AA was challenging because the execution environment did not provide the necessary primitives to construct these services.

2   Active Services

The goal of an AS is to offer a high-level network service that is focused and limited in what it does (i.e., only supports a limited set of methods), but yet can be customized to the particular needs of the application. Example AS services include reliable multicast

services, intelligent discard (congestion) services, mobility services, interest filtering, intrusion detection, or network management queries. In this paper, we consider a *concast* active service. Concast is roughly the inverse of multicast, merging packets as they travel from a set of senders to a receiver. Although we focus on the concast service, many of the issues that we address are typical of other ASes including:

Restricted Programming Model: An AS must "sandbox" the AA code thereby limiting it to do only what is allowed by that specific service.
AA Code Loading: Because an AS is itself a programming environment, it must be able to dynamically load an AA's code to customize the service.
Data and Control: An AS must handle both data and control packets. In addition to processing data packets, the AS typically implements a signalling (control) protocol by which AAs invoke the service and establish or tear down router state.
Simultaneous Use: Any number of AAs (flows), from multiple end-systems, may use the service simultaneously.
State Management: The AS maintains state for each AA (flow) using the service.
Scheduling: An AS may schedule service among the AAs based on the AS's policy.
Packet Demux: As data packets arrive, the AS must demultiplex packets (possibly after AS processing) to the appropriate AA for AA-specific processing. Note that this represents another level of demultiplexing beyond that offered by the EE.
Interrupt Processing: The AS may schedule timeouts to trigger events in the future (e.g., the transmission of packets at fixed time intervals).
Shared State: Some ASes may allow different AAs to communicate or share state.

Because these characteristics are similar to those of an EE, it is tempting to say an AS should just be an EE. However, there are subtle, but important, differences between ASes and EEs (in addition to the arguments given earlier against implementing an AS as an EE). First, an AS programming model is more focused and restrictive than an EE programming model. Second, an AS is not interested in re-implementing basic EE functionality like routing and forwarding. Third, an AS does not want to implement resource policies and mechanisms – things a NodeOS or EE must do. In short, an AS is not an EE. Instead, an AS should build on, and leverage, EE services.

3   The ASP EE

To explore the issues involved with implementing higher-level services (i.e., ASes), we implemented a *concast AS* as an AA running on the ASP EE. The ASP EE offers a Java-based programming environment that is designed to assist in the development and deployment of complex control-plane functionality such as signalling and network management. ASP implements the "strong EE model" [1] which means it essentially offers a user-level operating system to AAs. Although the underlying NodeOS and Java programming language provide operating system services, ASP tailors these services to its own goals and objectives. The primary enhancements are to the network I/O constructs. ASP uses *Netiod* to replace the socket interface with the active network input and output channel abstractions [6]. These channels effectively filter incoming and outgoing packets into streams. The interface between AAs and the ASP EE is called the

protocol programming interface (PPI), and allows AAs to access the active node's resources, such as storage, CPU cycles, and bandwidth, through the ASP EE. ASP also defines an interface to user applications (UA) on end systems. The UA can be thought of as 'adjacent' to the ASP EE and communicates with ASP using a TCP connection.

Being Java-based, ASP provides a certain amount of security that is inherited from the scope rules of the Java language. The ASP EE increases this protection by employing a customized Java Security Manager, which protects the EE and the JDK from aberrant or malfunctioning AAs. However, because each AA is assumed to be written by a single AA designer, there is no mechanism to protect certain parts of the AA from other parts of the AA (which we will see is needed by an AS).

ASP does not currently support in-band code loading, but rather supports an out-of-band loading mechanism. The ASP EE requires that active packets include an *AAspec*, which supplies a reference to the AA code to be run.

The ASP implementation executes each AA in a distinct process. Each process may contain multiple Java threads defined using the *AspThread Class*, which is akin to the Java thread class with some slight enhancements.

The ASP EE recognizes two methods for routing active packets: native (IP) connectivity and virtual (VNET) connectivity. Native IP connectivity allows AAs to use the IP protocol stack for routing. The ASP EE provides virtual connectivity via virtual links constructed using UDP/IP tunneling. The ASP EE's ability to provide routing functionality relieves the supported AAs from the responsibility of directly determining, maintaining, and using routing tables and protocols.

4   An Example Active Service: Concast

We begin with a brief overview of the *concast service*, and then describe our experiences implementing the service as an AA running on ASP.

4.1   The Concast Service Abstraction

Concast is roughly the inverse of IP multicast. Instead of a sender multicasting packets to a group of receivers, a *group of senders* send packets toward a single receiver. The concast service merges these packets together using a user-specified merge function [3].

A *concast flow* is uniquely identified by the pair (G, R) where R is the receiver's (unicast) IP address and G is the *concast group identifier*. G represents the set of senders that wish to communicate with R. Concast packets are ordinary IP datagrams with R in the destination field and G in the IP options field (in a *Concast ID* option). The packets delivered to R are a function of the packets sent by the members of G. Concast-capable routers intercept and divert for processing all packets that use the Concast ID option.

The concast abstraction allows applications to customize the mapping from sent messages to delivered message(s). This mapping is called the *merge specification*; it controls (1) the relationship between the payloads of sent and received datagrams, (2) the conditions of message forwarding, and (3) packet identification (i.e. which packets are merged together). The merge specification is defined in terms of four methods **getTag**, **merge**, **done**, and **buildMsg**. The concast framework allows users to supply the

definitions of these "merge-spec" functions using a mobile-code-language (currently Java). Once defined, a merge specification is injected into the network by the receiver and is pulled toward the senders as they join the group (using the *Concast Signalling Protocol (CSP)*). Additional details about the CSP protocol and merge specification can be found in [4, 5, 13].

Incoming packets are classified into flows based on (G, R). The flow's **getTag** function is applied to the packet to obtain a tag that identifies the equivalence class of packets to which the packet belongs. The **merge** function is then invoked to merge the current packet with the previous packets. The **done** predicate then determines whether the merge operation is complete. If so, **buildMsg** is invoked to construct an outgoing message from the merged state that is forwarded toward the receiver.

The two primary components of the concast service are the concast signalling protocol (CSP) and the merge processing framework. In our native (Linux) kernel implementation [5], we implemented these two components as UNIX processes: one process acted as the *Concast Signalling Protocol Daemon (CSPD)* and any number of processes executed *Merge Daemon (MergeD)* code (depending on the number of active concast flows). The CSPD handled all control messages (e.g., join and leave requests) and also "forked and exec'd" a MergeD process each time a new flow was initiated. Because each MergeD was a distinct process, MergeDs could not intentionally or accidentally affect one another.

4.2   Implementing Concast as an AS

As noted above, the concast service exhibits many of the characteristics typical of an active service(see Section 2). It needs to support multiple concast flows (i.e., must accept and process packets from multiple concast flows simultaneously). It must dynamically load code for each concast flow (i.e., the merge specification). It must handle both data traffic (packets to be merged) and control traffic (concast signalling messages to join/leave groups, pass merge specifications, or refresh state). For data traffic, it must ensure packets are delivered to, and processed by, the appropriate merging code. It also needs to focus (restrict) the programming model (i.e., only allow definition of the four merge-spec functions). Although it does not implement a scheduling policy for the processor, it does restrict the state that can be accessed and manipulated by the application code (i.e., merge spec). Finally, it needs to support interrupt processing for timer-triggered events (e.g., applying periodic processing or forwarding packets after fixed time intervals).

Figure 3 illustrates our design for the concast AS, and shows how it fits into the overall active network framework.

The first challenge arises from the need to support multiple concast flows (groups) using the service simultaneously (i.e., the AS must support multiple MergeDs executing simultaneously). Unfortunately, ASP provides no mechanism by which an AA can "fork and exec" a new AA to perform parallel MergeD processing.

To achieve the desired parallelism, we implement the CSPD and all MergeDs as distinct *ASP threads* inside a single AA. Assigning each merge specification to a different ASP thread allows us to maintain clean separation of the (independent) MergeDs from one another. Each thread maintains merge state specific to that flow. It also offers the
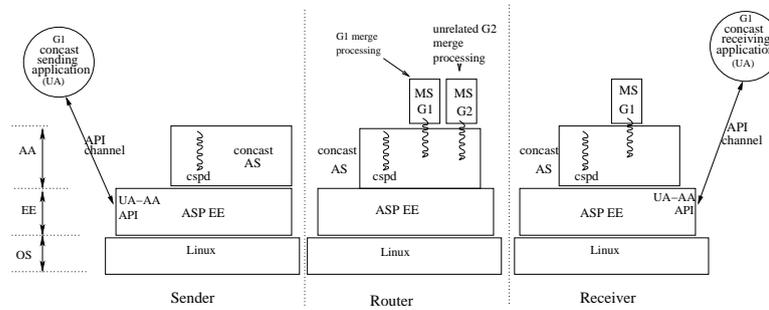
Fig. 3. The Concast Service implemented as an AA on ASP.

opportunity to execute the application's customization code (i.e. merge spec) in parallel. Although parallel execution is not necessarily important for concast merge specs, it is important for other active services that do not always process packets to completion. The alternative, using a single thread to execute the customized code from all applications, is possible, but complicates the code and can produce unnecessary blocking, particularly when a packet cannot be processed to completion.

Given multiple merge specifications executing simultaneously in distinct ASP threads, a related issue is the problem of getting incoming concast packets to the appropriate MergeD (i.e., ASP thread). As noted earlier, ASP allows an AA to establish one or more in-channels (classifiers) that define the set of packets the AA should receive. Because ASP wants to deliver packets to the AA in the order that the packets were received (across all in-channels), ASP delivers all packets to the main ASP thread (regardless of the in-channel that the packet is associated with). In other words, ASP does not record which thread opened the in-channel and thus cannot deliver packets for a channel directly to the thread that requested the packets.

Because ASP delivers all packets directly to the main thread, we had to implement the ability to demultiplex packets to the appropriate threads (i.e., MergeDs). To do this, we used the main ASP thread as a "packet dispatcher", examining each packet's receiver address R and group id G, and demultiplexing the packet to the appropriate MergeD thread. If the packet is a CSP (control) packet rather than a data packet, the main thread immediately applies the CSPD processing, possibly creating or terminating a thread (i.e., a MergeD) in response to a CSP join or leave message. Ideally, the underlying EE would have offered the ability to assign in-channels to threads rather than AAs (thereby avoiding this demultiplexing overhead).

This raises the next issue: interthread communication. Interthread communication was implemented using shared java monitor queue objects between the dispatcher thread and the MergeD threads. By wrapping the queue in a java monitor object, we were able to ensure mutually exclusive access to the queue. The dispatcher thread read the packet, identified the appropriate queue object, and then used the monitor to lock and append the packet to the queue. It then signalled to inform the MergeD thread that a packet was available. In this case, the base language offered the support we needed. However, if the

base language did not offer this support, we would have had to implement our own IPC mechanism.

Note that our design suffers from the same protection and security issues that plague EE's that support multithreading [14]. Systems that support threads typically allow all threads to access the same data (i.e., shared access). As a result, the merge-spec for one flow could conceivably interfere with, kill off, or otherwise disrupt the MergeD of another flow. ASP actually protects AAs from one another by executing each in its own JVM (i.e., its own virtual memory process). The ANTS EE [14] faces a similar problem, namely ensuring that data deposited by packets of one protocol cannot be accessed by packets in another protocol. In Java, accessing objects requires a reference to the object, and such references can only be obtained through legitimate channels (i.e. it is not supposed to be possible to convert arbitrary data to references). ANTS arranges things so that only authorized code can obtain a reference to an object, because the reference is tied to a cryptographic hash of the code itself! Thus, in ANTS, objects are secure from tampering by other flows. In the case of our concast AS, the MergeD code executes in the context of an ASP thread that does not know about, interact with, or exchange references with any other threads; consequently, data and packets referenced by one thread are not accessible to another thread, thereby guaranteeing protection between threads.

One example where the ASP EE restricts the base-language primitive in a way that hinders the development of an AS is the ASP class loader. Rather than reinvent our own class loader, we wanted to utilize the ASP class loader. Although the ASP class loader greatly simplified our implementation, it also limits our implementation in some minor, but important, ways. In particular, to prevent an AA from dynamically loading code from anywhere, the ASP class loader requires that all loadable code (including the AA code) reside in one of a set of pre-specified directories that are associated with the AA. However, in the case of an AS, the code will reside at a location of the application's choosing (i.e., an arbitrary location). Because ASP restricts the location where code can be loaded from, its loader is not well suited for our application. To get around this problem, we enabled applications to deposit their code at the URL for the concast AS. This is clearly not an elegant or secure solution.

A related problem is that of restricting the programming environment offered by an AS. ASP uses a customized Java Security Manager to protect ASP from the AAs. However, the goal is security/safety rather than restricting the programming model – almost all Java functionality is available to the AA. Although we have not implemented one, our concast AS could also develop its own specialized security manager to disable certain functionality. Although this would limit the programming model, it may not be powerful enough to define the specialized programming models offered by certain ASes.

One of the biggest problems we faced was implementing timeouts. The concast programming environment supports the notion of timeouts to trigger periodic processing of data or transmissions of already merged data. This feature is used, for example, in the audio merge example described in Section 5. The primary problem we encountered was the inaccuracy of the timeout mechanism. Although we still have not completely identified all the contributors to the problem, one of the major factors was the layering

that caused the timing errors to accumulate. In concast, timeouts are implemented using an ordered per-merge-spec timeout list. Each time the merge framework processes a message it checks for any pending timeouts, processes those, and then issues a timed wait for the next packet (based on the next timeout to occur). Like most timeout mechanisms, this implementation does not guarantee that the timeout processing will occur precisely at the point of the timeout, but rather "soon after" the timeout. The concast timeout mechanism itself relies on the underlying timeout mechanism provided by Java (i.e., a timed wait). The Java mechanism itself relies on the timing mechanisms of the underlying operating systems. Because each layer depends on the "imprecise" timeout mechanism of the layer below it, the accumulation of errors often lead to unacceptable accuracy at the AS level.

We experienced variance as high as 10ms which was beyond the tolerance of our audio example. Even if a single error of a few milliseconds could be tolerated, repeated inaccuracy over time led to an accumulation of errors that were unacceptable. Although our measurements show that a significant portion of the inaccuracy can be attributed to the timed waits in Java, it appears that some of the inaccuracy accrued from thread switching and packet demux. If the AS can be implemented with a single thread, we expect the accuracy would improve, but will still suffer the effects of layering.

In short, some aspects of ASP made it easy to offer the necessary services and features, while other aspects were not well-suited as ASP is currently designed. However, with feedback about the characteristics needed by the average AS, EE's could be designed to offer the appropriate support so that implementing an AS as an AA is both viable and attractive.

5   Experimental Results

To evaluate the performance aspect of an AS implemented as an AA, we developed an audio merge application that employs our concast AS service to do the merging. In what follows, we describe the audio merging application, our experimental environment and setup, and results obtained from a running system.

5.1   An Audio Merge Example

To test our concast AS and measure its performance characteristics, we constructed an audio application that merges together multiple audio flows to form a single "combined" audio flow that is delivered to a (concast) receiver. This type of audio merging/mixing might be used in conference calls or teleconferencing, distributed music performances, or it might be used to merge audio from distributed sensors used for audio surveillance.

The basic idea behind our approach is that each sender transmits a u-law encoded audio stream at a fixed rate (i.e., 64 kbps in our case) to a receiver (i.e., the merge destination). The goal is to combine the audio streams and play them as a single audio stream at the receiver. The receiver creates the concast flow by installing an "audio merge specification". The audio merge specification executes at each router combining an audio packet from each of the incoming flows to create a single outgoing packet. In our case, merging is achieved via a simple function that first maps u-law encoded

audio samples to a-law encoded samples. It then combines the samples by adding a-law values from different flows together and truncating the results if the result exceeds the maximum allowable value. Finally, it maps the resulting a-law sample back to a u-law encoded sample. The merge function also includes timeout processing. The intent is that as packets arrive, the merge function merges the incoming packet into the state accumulated thus far. At regularly scheduled 125 ms intervals, the node packages the accumulated state and forwards it toward the next downstream router. Senders transmit a 64 Kbps audio stream, sending a packet every 125 ms where each packet contains 1000 eight-bit samples. The intermediate nodes are impervious to packet loss; they blindly forward whatever they accumulate during the 125 ms time period. If no packets were received, the node forwards nothing during that interval.

Because of the difficulty realizing precise timeouts in the AS, we implemented a version of the MergeD that buffered up to 3 packets from each upstream neighbor. Every 125 ms the MergeD removed the first buffered packet from the queue for each upstream neighbor, merged them, and forwarded the result. Normally, if a node received two consecutive packets from the same sender during a single 125 ms interval, it would discard the second packet (because we do not want to merge together packets from the same sender). By employing a small amount of buffering at each node, the merge function became much less susceptible to packet drops resulting from timing errors.

5.2   Experimental Setup

We tested our concast AS and audio merge applications using the topology shown in Figure 4. We implemented the running system and topology using the EMULAB system at the University of Utah [8]. We used pre-recorded audio files as the audio input to the senders, and the receiver wrote an audio output file.[1] The last-hop link in the topology
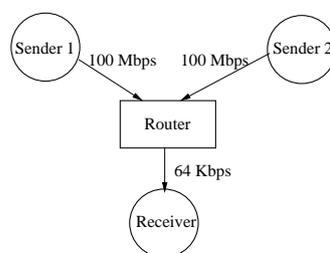
Fig. 4. Topology of our test network.

was intentionally designed to be a bottleneck link; the 64 Kbps link can support at most one audio flow at a time. Our experiments consisted of two senders, one router and one receiver.[2] Each node ran the Linux operating system, ASP, and our concast AS.

---

[1] We also tested a live playout.

[2] We have tested with more elaborate topologies and more senders, but two senders is sufficient to illustrate the AS performance.

Senders and receivers use the *User Application (UA)* and the *UA-AA* API of ASP to send active packets to and receive active packets from the ASP AA. ASP nodes communicated using ASP's *VNET* network interface. A concast flow is created when a concast receiving application sends a CREATE request to a local concast AS using the UA-AA API. The CREATE request specifies the concast address and merge specification and indicates the receiver's willingness to accept data sent to this concast address. Subsequently, the concast sending applications can join this concast flow by sending a JOIN request to their local concast AA (using the UA-AA API). This triggers the concast signalling protocol which "pulls down" the merge spec and loads it via the ASP class loader.

As a point of comparison, we also implemented a unicast version of the merge processing. In the unicast case, the senders transmitted (independent) unicast flows to the receiver; the receiver then merged the packets together before playout. Because the unicast flows were not merged until they reached the receiver, their combined load exceeded the bandwidth of the last-hop link.

5.3   Performance Results

Figure 5a compares the number of packet drops that occur under the the concast implementation (with a buffer size of 3 packets per sender) and the unicast implementation. The more packets dropped, the worse the sound quality. One sender transmitted melodic background music while the other sender transmitted someone talking. In the unicast implementation, the music was choppy and the speaker could not be understood. The concast implementation with buffering completely eliminates all packet drops, particularly the drops caused by the timing inaccuracies of the AS. The resulting sound quality was very good, because there are almost no packet drops.
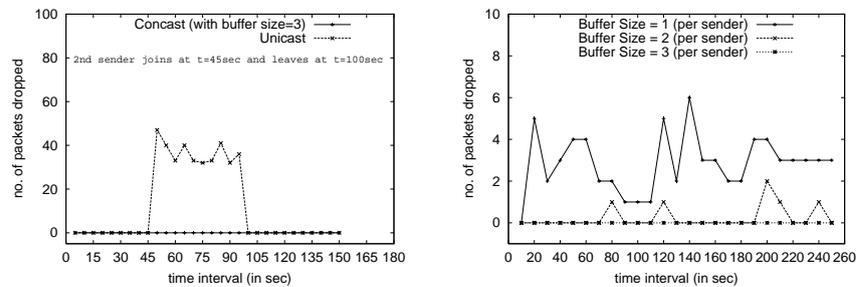
Fig. 5. (a) Packet drops for Concast vs. Unicast. (b) Packet drops for Concast with various buffer sizes.

Figure 5b illustrates the effect buffering has on the performance of the concast AS. Because timing inaccuracies can lead to "bursts" of up to 3 packets (i.e., from the same sender arriving in the same 125 ms interval), buffer sizes of only 1 or 2 packets will drop some packets. These packet losses produced a periodic "ticking" sound in the audio. A

buffer size of 3 packets per sender eliminated all packet drops in our experiment and produced good sound quality.

6   Related Work

Past work in active nets [2] has focused largely on the NodeOS and EE levels. EEs such as ANTS [14] and ASP [1] are Java-based environments each leveraging the Java language in slightly different ways. Had we selected ANTS as the development environment as opposed to ASP, most of our approach would have remained the same given the fact that they are both Java-based. Other EEs, such as PLAN [9, 12], could also have been used. Although we have relatively little hands-on experience with the PLAN EE, we expect that it would have been more helpful with defining the restricted programming model desired by the AS. The CANEs [7] EE offers a more restricted programming model and may have presented some special challenges to implementing an AS as an AA.

Relatively few active applications have been implemented. The few that have been implemented focus on a total solution to a problem such as video processing [11], distributed interest filtering [15], and network management [10]. Each of these executes as a complete AA. Concast is one of the first higher-level AAs that actually offers a service and programming environment to other applications.

7   Conclusions

In this paper we addressed the issue of implementing higher-level active network services; services that are more specialized than EEs, but yet are programmable like EEs. We call these middleware services *active services* (AS). We outlined three methods for implementing active services: EEs, AAs, and libraries. Given the drawbacks of EEs and libraries, we decide to explore the approach in which an AS is implemented as an AA.

To understand the issues associated with this approach, we implemented the *concast* AS (as an AA) in the ASP execution environment. We described the characteristics of an active service, and showed which characteristics were straightforward to implement in ASP and which ones presented problems. Features such as multitasking were straightforward to implement, while other features, such as packet demultiplexing and code loading required re-implementing (or adapting) these services at the AS level. The characteristic that posed the biggest challenge was timer management. We presented experimental results from an audio application that show the performance of an AS can be acceptable for realtime packet processing.

In summary, we've pointed to EE features that were well suited for supporting active services, and other features that could be designed better. Our experience indicates that implementing active services as AAs is a viable approach, both from an ease-of-implementation standpoint and a performance standpoint. As we gain more experience with active services, time will tell if AAs are the correct approach for implementing active services.

References

[1] Robert Braden, Bob Lindell, Steven Berson, and Ted Faber. The ASP EE: An Active Network Execution Environment. In *DARPA Active Network Conference and Exposition (DANCE) 2002*, May 2002.

[2] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10):72–78, October 1998.

[3] K. Calvert, J. Griffioen, B. Mullins, A. Sehgal, and S. Wen. Concast: Design and implementation of an active network service. *IEEE Journal on Selected Areas of Communications*, 19(3):426–437, March 2001.

[4] 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.

[5] K. Calvert, J. Griffioen, A. Sehgal, and S. Wen. Implementing a concast service. In *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, September 1999.

[6] Kenneth L. Calvert. An Architectural Framework for Active Networks, 2001. DARPA Active Nets Document, http://protocols.netlab.uky.edu/ calvert.

[7] Kenneth L. Calvert and Ellen W. Zegura. Composable Active Network Elements Project. `http://www.cc.gatech.edu/projects/canes/`.

[8] Flux Group. Emulab Network Testbed. Computer Systems Lab, University of Utah, http://www.emulab.net/.

[9] Michael Hicks, Pankaj Kakkar, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the International Conference on Functional Programming*, 1998.

[10] A. Jackson, J. Sterbenz, M. Condell, and R. Hain. Active Network Monitoring and Control: The SENCOMM Architecture and Implementation. In *DARPA Active Networks Conference and Exposition*, pages 379–393, San Francisco, May 2002.

[11] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner. An Active Router Architecture for Multicast Video Distribution. In *IEEE INFOCOM*, Tel-Aviv, Israel, March 2000.

[12] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical Programmable Packets. In *IEEE INFOCOM*, Anchorage, AK, April 2001.

[13] Amit Seghal, Kenneth L. Calvert, and James Griffioen. A Flexible Concast-based Grouping Service. In *Proceedings of the International Working Conference on Active Networks (IWAN) 2002*, December 2002.

[14] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, 1998.

[15] S. Zabele, M. Dorsch, Z. Ge, P. Ji, M. Keaton, J. Kurose, and D. Towsley. SANDS: Specialized Active Networking for Distributed Simulation. In *DARPA Active Networks Conference and Exposition*, pages 356–365, San Francisco, May 2002.