Bowman and CANEs: Implementation of an Active Network*

S. Merugu† S. Bhattacharjee♯ Y. Chae† M. Sanders† K. Calvert ‡ E. Zegura †

†College of Computing, Georgia Tech, Atlanta, GA
♯Dept. of Computer Science, Univ. of Maryland, College Park, MD
‡Dept. of Computer Science, Univ. of Kentucky, Lexington, KY

Abstract

In parallel with active networks research, efforts have been underway to define and standardize an architectural framework. The framework divides the functionality of an active network node into two components: an Execution Environment (EE) defines a programming interface that allows users to control the active network, a NodeOS defines a set of basic functions to access and manage the resources of the active node. EEs use the abstractions provided by the NodeOS to build a virtual machine made available to users. Feedback from implementation efforts is critical for refining the standards being developed.

We are building an active network comprised of the CANEs execution environment and the Bowman NodeOS. Bowman is constructed by layering active-network-specific operating system functionality on top of a standard host operating system. The host operating system provides low level mechanisms; Bowman provides a channel communication abstraction, an a-flow computation abstraction and a state-store memory abstraction, along with an extension mechanism to enrich the functionality. The CANEs EE provides a composition framework for active services based on customizing a generic underlying program by injecting code to run in specific points called slots. This paper reports on our experience in implementing Bowman, instantiating CANEs on top of Bowman, and developing applications within CANEs.

1 Introduction

In parallel with research experience in active networks, efforts have been underway to define and standardize an architectural framework [3, 2]. In the draft framework, the functionality of an active network node is divided between Execution Environments (EEs) and the Node Operating System (NodeOS) [4] as shown in Figure 1. An EE defines a programming interface that allows users to control the active network; a NodeOS defines a set of basic functions to access and manage the resources of the active node. EEs use the abstractions provided by the NodeOS to build a virtual machine made available to users.

The early development of EEs (e.g., ANTS [9], PLAN [5]) occurred prior to standardization and development efforts for a NodeOS, thus the early EEs made use of traditional

*This research is supported by DARPA under contract number N66001-97-C-8512.

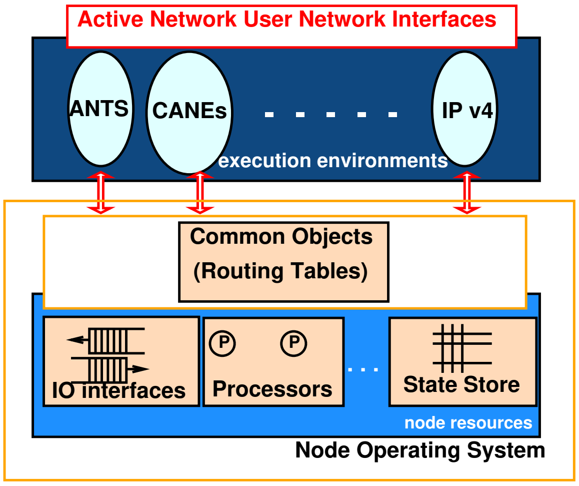


Figure 1: Components of an active network node.

machine operating systems for resource access and management. Experience in developing EEs on top of NodeOS's is, of course, critical for refining the standards being developed. NodeOS projects (e.g., Janos [6], Joust [7]) are developing implementations that complement our experience. In particular, these projects have largely focused on supporting Java and ANTS, while leveraging existing efforts in high-performance operating systems.

We are building an active network comprised of the CANEs execution environment and the Bowman NodeOS. Bowman is constructed by layering active-network-specific operating system functionality on top of a standard host operating system. The host operating system provides low level mechanisms; Bowman provides a channel communication abstraction, an a-flow computation abstraction and a state-store memory abstraction, along with an extension mechanism to enrich the functionality. The CANEs EE provides a composition framework for active services based on customizing a generic underlying program by injecting code to run in specific points called slots. This paper reports on our experience in implementing Bowman (Section 2, instantiating CANEs on top of Bowman (Section 3), and developing applications within CANEs (Section 4). We conclude in Section 5 with a discussion of lessons learned and open problems.

2 Bowman

Bowman provides three basic abstractions to support active network functionality: *channels*, *a-flows* and *state-store*. In addition to these three abstractions, Bowman has an *extension* mechanism that is similar to loadable modules in traditional operating systems. Using the extension mechanism, additional components of a Bowman node can be loaded dynamically at run-time. This section discusses the basic abstractions, related extensions, packet processing path and Bowman performance.

Channels Channels represent the communication resource exported by Bowman. They are communication end-points that support sending and receiving packets via an extensible set of protocols. Bowman exports a set of functions enabling users (i.e. EEs) to create, destroy, query and communicate over channels that implement traditional protocols (TCP, UDP, IP, etc.) in various configurations. In addition, channels in Bowman can include other forms of processing such as compression, forward error correction on the packets.

For packets that do not require per-flow processing, Bowman provides *cut-through channels*, i.e., paths through the Bowman packet processing cycle that do not incur the overhead of multi-threaded processing.

Globally, Bowman implements a network-wide architecture by providing system support for multiple simultaneous *abstract topologies*, i.e., overlay network abstractions that can be used to implement virtual networks. Abstract topologies are built from *links*, which are an association of two of more channels.

A-flows The a-flow is a primary abstraction for computation in Bowman. A-flows encapsulate processing contexts and user state. Each a-flow consists of at least one thread and executes on behalf of an identified principal. Bowman provides a set of functions to create, destroy and run a-flows.

Efficient packet classification to identify flows is an essential part of flow-specific processing. The complexity of packet classification algorithms depend on (1) the type of matching required (first match, longest prefix match, or *best* match based on attributes of matched fields), and (2) the number of fields and types of headers being matched.

Since the identification of a packet is solely based on the pattern of bits contained (possibly, at any location) in the packet, we have implemented a packet classifier that can match on (1) an arbitray number of fields, and (2) each field at an arbitrary offset in the packet. The classifier can be configured (as a matter of node-wide policy) to operate in one of the three modes: return the first match, return all matches, and return the best match(es) according to a cost associated with each field in the classification rule-base.

The Bowman packet classification algorithm does not depend on the protocols used to frame the contents of the packet. But in order to specify a meaningful (readable) pattern of bits to be used to match incoming packets, we can first teach the classifier about a protocol (e.g., Ethernet) and its keywords (e.g., `src`, `dst`, `proto`) using a small language. We can then use these keywords to encode a pattern that will be used to match the packets.

Bowman also provides a fast timeout routine that supports multiple timers per-a-flow. Using the Bowman timeout routine, we have implemented a Bowman extension for threaded per-a-flow timers. Each a-flow contains one timer thread that executes call-back functions when specific timers expire. The resolution of the timers and the number of outstanding timers are configurable parameters.

State-store The state-store provides a mechanism for a-flows to store and retrieve state that is indexed by a unique key. The Bowman state-store interface provides 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 sharing data between a-flows without sharing program variables.

Packet processing path Figure 2 shows a schematic of the Bowman packet processing path. The figure shows the demarcation between Bowman system code and user code that runs within particular a-flows. Within the underlying Bowman system, there are (at least) three processing threads active at all times: a packet input processing thread, a packet output processing thread and a system timer thread. Depending on the input

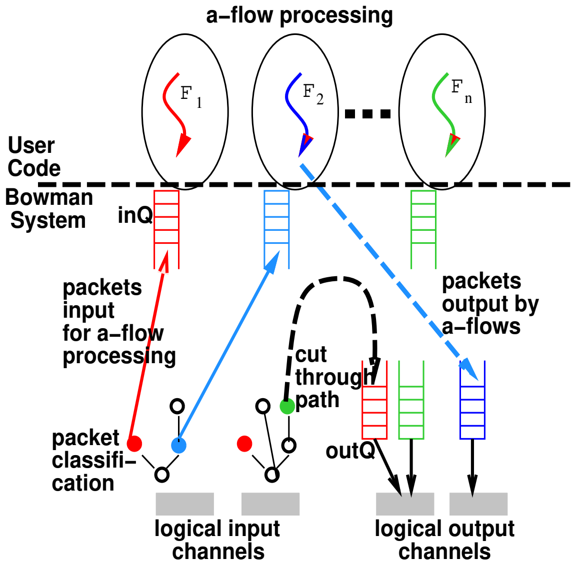


Figure 2: Bowman packet processing path.

and output scheduling algorithms, it is possible for multiple input and output threads to be active within the Bowman.

Packets received on a physical interface undergo a classification step that identifies a set of channels on which the received packet should be processed. In this manner, incoming packets are demultiplexed to specific channel(s) where they undergo channel-specific processing. Once channel input processing completes, the packet is considered to have "arrived" on the abstract link associated with the channel. At this stage, the packet undergoes a second classification step that identifies the further processing (a-flow or cut-through) needed by the packet.

A-flow processing is determined by the code specified when the a-flow is created. Using the Bowman extension mechanism, a-flow code can be dynamically introduced into a node; this mechanism is also used for dynamic code loading (as is required for some active network execution environments).

Bowman performance Our goal in developing Bowman is to leverage the facilities available in traditional operating systems and concentrate on the active-network-specific aspects of a NodeOS. However, we also want to maintain reasonable performance. Bowman has several performance-enhancement features. First, the Bowman implementation is multi-processor capable; on multi-processor machines, Bowman threads may execute concurrently and some thread context-switch times are eliminated. Second, on the Solaris platform, Bowman can be configured to run in real-time mode using the Solaris Real-Time Extensions.[1] With real time extensions, Bowman can deliver high throughput and low delay for packets traversing through a-flows. To leverage the performance advantages of a multi-processor multi-threaded implementation, we have implemented the a-flow input and output queues so that, for single-reader single writer queues, there is no locking (or blocking) unless the queue is empty.

[1]See the `priocntl(2)` system call on SunOS 5.5+. An effort is underway to port Bowman real-time extensions to RT-Linux.

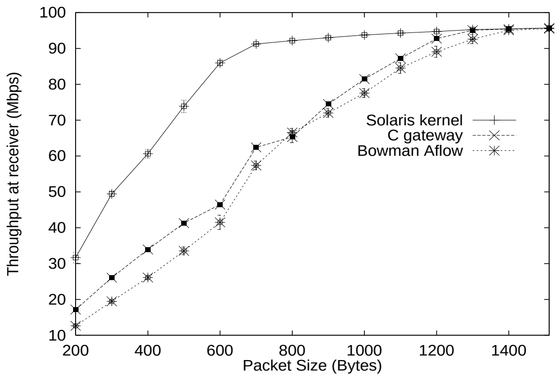


Figure 3: Sustained throughput

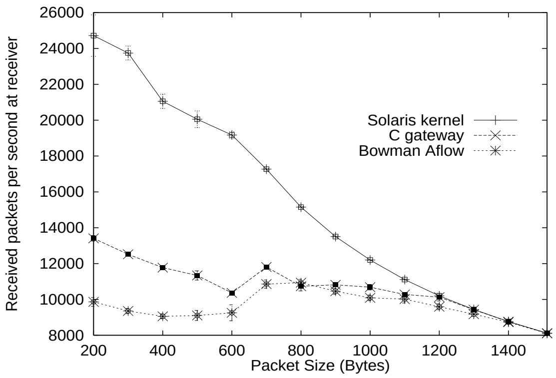


Figure 4: Sustained packet processing rate

We have evaluated the performance of Bowman in a variety of configurations. We include an example result of base forwarding performance here; more detail can be found in [8]. These experiments were performed on a three-node testbed configured in a straight line topology. The interior node executed Bowman. The edge nodes were directly connected to separate 100 Mbps interfaces to the Bowman node. We compare the performance of the Bowman packet processing path with that of the forwarding performance of the Solaris kernel and a gateway program written in C. The single threaded C gateway program uses `socket` calls to read and immediately write out UDP segments. In each case, the sending node transmits UDP segments of a specified size to the destination node. The routing table of one of the edge nodes was configured to use the Bowman node as its next hop to the other edge node in our testbed. The Bowman topology was configured with two bidirectional UDP channels — from the edge nodes to the interior node. (For the C gateway and Bowman forwarding experiments, the Solaris kernel forwarding was turned off).

Figure 3 shows the sustained lossless throughput through Bowman, the C gateway, and the Solaris kernel as measured at the receiver. Figure 4 shows the raw packet rate (in packets/second) at the receiver for the same experiment. It is clear from the figures that the system call, context switch and data copy overheads for small packet sizes precludes both the C gateway and Bowman from sustaining high throughput. However, as packet sizes increase, the context switch and system call dispatch overheads are amortized as more data traverses the system and user space boundary for each system call. Bowman is able to saturate the 100 Mbps ethernet for packet sizes of 1400 bytes or more; Bowman performance is within 5% of kernel forwarding for packet sizes greater than 1200 bytes.

3 CANEs

We have used the Bowman NodeOS as the platform for implementing the CANEs execution environment. This section gives an overview of CANEs, then describes key aspects of the implementation.

Overview The slot processing model used in CANEs EE comprises two parts: a fixed part (*underlying program*) that represents the uniform processing applied on every packet, and a variable part (*injected program*) that represents user-specific functionality on the packets. The specific points in the underlying program where the injected program may be executed are called *slots*. Composition of services in a CANEs computation is achieved in two steps. First, an underlying program that provides a basic service (e.g. forwarding) can be selected from amongst those offered by an active node. In the second step of composition, a set of injected programs are selected to customize the underlying program. These injected programs can be available at the active node, or can potentially be downloaded from a remote site. Each injected program is bound to one or more processing slots.

Figure 5 illustrates a generalized forwarding function (GFF) underlying program for CANEs that supports forwarding to a small number of destination addresses. The GFF has four parameters. Two are required: a source address S, a list of addresses A. Two are optional: a forwarding table identifier R and a selection function M used to match addresses with forwarding table entries. If the optional parameters are not supplied, defaults will be used. The GFF also exports slots where the user may bind injected programs. Each slot has a default behavior, indicated in the square brackets and used if the user does not supply an alternative. For example, Slot 2 is reached if the list of output interfaces turns out to be null. By default, no error message is generated; the user may choose, for example, to send an error message to the source. The other slots provide opportunities to control action taken upon packet receipt and further processing (e.g., sending notification, incoming and outgoing topology information to the source, application of per-interface policy, flow-specific congestion control etc.).

Implementation Figure 6 shows a thread-level abstraction of our CANEs implementation over Bowman. Upon startup, the CANEs EE spawns the control a-flow that contains a thread for handling signaling messages. Each user's computation is executed in its own a-flow and can spawn multiple independent threads that share an address space. The CANEs EE itself provides a library that implements the CANEs API and resides in the system as a single EE a-flow that contains a set of threads to handle housekeeping chores (like signaling and timers).

The CANEs signaling messages are written in the CANEs User Interface (CUI) language and are sent to a well-known port. This message consists of two parts: computation and communication. The computation part defines a directed acyclic graph (DAG). The root of the DAG is an underlying program. Each child node corresponds to an injected program bound to a slot in its parent. The arcs in the DAG identify the slot to which the child is bound. Each node contains enough information for the CANEs EE to fetch the code required to execute the computation DAG. The communication part of the CUI message identifies a routing schema for the user. Further, the user can specify a set of packet classifier rules that are used to select incoming packets that the flow acts on. This communication part provides enough information for the CANEs EE to establish the input and output channels required for the user's computation.

When the underlying program is dynamically loaded, the default entry function "_entry" is executed. This function publicizes the slots exported by the underlying program. Binding of injected programs to the slots is done while parsing the computation part of the CUI signaling message. Invoking a slot is analogous to raising a named event. In this metaphor, all injected programs bound to a slot are handlers for the named event and are executed concurrently when the event is raised.

The CANEs EE provides a variable sharing mechanism that is used by injected programs to communicate with underlying programs. Underlying programs declare shared

```
while (Read next packet) {
    Parse packet to obtain S, A, [R, M]
    ⟨Slot 0:[null]⟩
                    ; Packet Arrival Slot; Possible uses: route trace, caching
    Outputlist := ⊥
                    ; Initialize the outgoing set of interfaces to null
    For each address a in A: {
        Let interface i := Lookup(a, R, M)
                    ; Look up the interface for each address a
        ⟨Slot 1:[Add i to Outputlist]⟩
                    ; Code bound to this slot may implement policy
                    ; to reject certain interfaces
        }
    if Outputlist = ()
        then ⟨Slot 2:[null]⟩; abort;
                    ; When no output interfaces were found, the code
                    ; bound to slot 2 can send errors to source
    For each unique i in Outputlist:{
                    ; Packet will be dispatched on each outgoing interface
        Create a copy D of the packet,
                    with A' = {a : (i, a) ∈ Outputlist}
                    ; The new destination address set contains only addresses
                    ; that can be reached via interface i
        if i is congested
            then ⟨Slot 3:[discard]⟩
                    ; Congestion control algorithms may be bound to slot 3...
        else ⟨Slot 4:[null]⟩
                    ; ...alternate schedule policies to slot 4
            enqueue D for i.
                    ; Eventually, the packet is transmitted on interface i
        }
    }
```

Figure 5: Example CANEs underlying program.

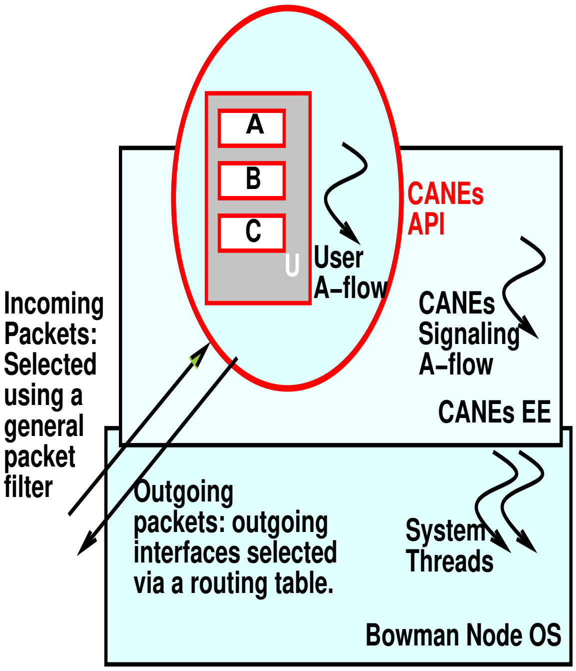


Figure 6: A high level view of processing at an active node running the CANEs EE over the Bowman Node OS. User flows consist of an underlying program (U in this figure) and a set of injected programs (A, B, and C). This figure also shows a nominal thread level structure of our implementation.

variables and *export* them (make them eligible for sharing). Injected programs declare *references* to shared variables (the space for the actual data for references must be declared by the underlying program). Further, injected programs *import* shared variables exported by underlying programs (this finally creates the binding between the shared and the references variables).

4 Applications

We are in the process of accumulating experience with implementation of applications in CANEs. In this section we describe two moderately complex applications.

Active error recovery

Active Error Recovery(AER) [1] is a joint effort between TASC and the University of Massachusetts to develop a reliable multicast framework using active processing. We have worked with researchers from TASC and University of Massachusetts on an implementation of AER in CANEs.

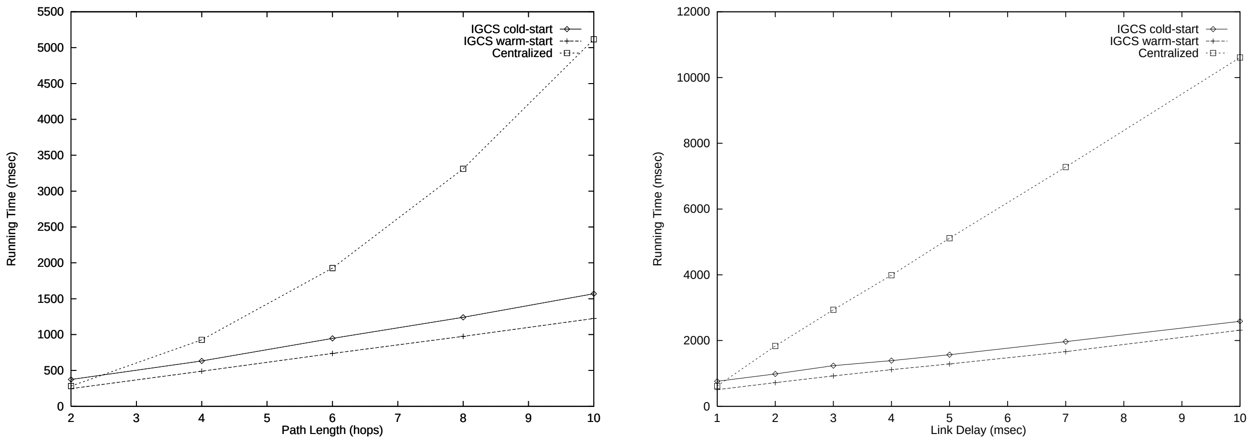
AER makes use of a repair server, residing within the network, to cache packets, respond to retransmission requests, suppress redundant NAKs from receivers, and detect gaps in sequence numbers (indicating lost packets). AER also includes protocols to calculate round trip times and dynamically select a worst receiver to handle sliding-window-based flow control.

We have implemented AER in CANEs using two underlying programs, a multicast forwarding function for sending data, NAKs and source path messages, and the generalized forwarding function for calculating the round trip time and monitoring the congestion status. We make considerable use of the system timers to retransmit and suppress NAKs, to uninstall stale state associated with a flow, and to calculate round trip times.

Iterative gather-compute-scatter

The Iterative Gather-Compute-Scatter (IGCS) distributed computation model provides a mechanism to query and synthesize network state [10]. Such a mechanism is potentially useful for a range of applications that are sensitive to network topology (e.g., placement of an AER repair server). IGCS programs repeat a gather, compute, scatter cycle until a given condition is satisfied. During an iteration, a set of messages are collected during the gather phase. Once a specific set of messages have been collected, the compute phase commences. The inputs to the computation are the set of collected messages, the node and link attributes, and the state store for this computation. The compute phase can produce a single message (of a fixed IGCS message type) that is then "scattered", i.e. transmitted to a set of destinations. IGCS programs can retain state at a node while they are active. All state is lost after the last iteration of the IGCS computation.

The implementation of IGCS consists of an IGCS daemon, an IGCS underlying program and a set of injected programs. The daemon (implemented as a Bowman extension) receives signaling messages and initiates new computations at network nodes. During the initiation of a computation, the underlying and compute slot programs are loaded onto the local node via the Bowman code loading mechanism. The compute slot programs are bound to the proper compute phases. The data part of the signaling message is stored into the local state-store so that the information can be retrieved by the IGCS computation.



(a) Effect of path length

(b) Effect of link delay

Figure 7: Running time with varying link delay and path length

Figure 7 shows the running time of an IGCS application to find the minimum bandwidth on a path. We compare the IGCS implementation to a centralized scheme that must query each node on the path sequentially. In the plot on the left, we vary the length of the path while keeping link delay constant at 5 msec. In the plot on the right, we fix the path length at 10 hops and vary the link delay. For the IGCS algorithm, we show results for both cold-start (i.e., code must be loaded) and warm-start (i.e., code is resident in the cache). This experiment has two salient points. First, the IGCS scheme has a significant performance advantage over a non-active scheme when the number of hops exceeds two. Second, the overhead of code-loading in this example is modest; the cold-start overhead is about 35 msec, while the warm-start overhead is about 5 msec.

5 Discussion

We briefly comment on lessons learned in implementation and on current areas of work. Bowman is intended to be a "shim" layer over a traditional operating system. However, providing useful active networking functionality requires access to low level resources, thus Bowman cannot be insulated from low level details of the OS. Said another way, the "active" part of an active OS (especially one with decent performance) must be reasonably OS-savvy.

We originally designed CANEs so that all slots were raised due to packet arrivals. However, AER (and likely other protocols) require activity that is driven by timers, rather than packet arrivals. We added the ability to raise slots in timer handlers as a result of experience with AER implementation.

We are currently adding to our implementation. Areas of near-term work include better security and protection mechanisms, more sophisticated output queueing schemes, and scalable protocols for virtual topology instantiation.

References

[1] Active Error Recovery (AER). http://www.tascnets.com/panama/AER.

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

[3] Kenneth L. Calvert (Editor). Architectural Framework for Active Networks. DARPA AN Working Group Draft, 1998.

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

[5] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. Available at `www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`.

[6] Janos: A Java-based Active Network Operating System. http://www.cs.utah.edu/projects/flux/janos/summary.html.

[7] Joust: A Java OS in Scout. http://www.cs.princeton.edu/nsg/joust.html.

[8] S.Merugu, S.Bhattacharjee, E.Zegura, and K.Calvert. Bowman: A Node OS for Active Networks. Submitted to IEEE Infocom 2000.

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

[10] Y.Chae, S.Merugu, E.Zegura, and S.Bhattacharjee. Exposing the network: Support for topology-sensitive applications. Submitted to IEEE OpenArch 2000.