Bowman: A Node OS for Active Networks

S. Merugu † S. Bhattacharjee‡ E. Zegura† K. Calvert♯

† College of Computing, Georgia Tech., Atlanta, GA. {merugu, ewz}@cc.gatech.edu

‡ Dept. of Computer Science, Univ. of Maryland, College Park, MD. bobby@cs.umd.edu

♯ Dept. of Computer Science, Univ. of Kentucky, Lexington, KY. calvert@cs.uky.edu

Abstract—**Bowman is an extensible platform for active networking: it layers active-networking functionality in user-space software over variants of the System V UNIX operating system. The packet processing path implemented in Bowman incorporates an efficient and flexible packet classification algorithm, supports multi-threaded per-flow processing, and utilizes real-time processor scheduling to achieve deterministic performance in user-space. In this paper we describe the design and implementation of Bowman; discuss the support that Bowman provides for implementing execution environments for active networking; discuss the network-level architecture of Bowman that can be used to implement virtual networks; and present performance data. Bowman is able to sustain 100 Mbps throughput while forwarding IP packets over fast Ethernets.**

I. Introduction

Active networks provide a *programmable* user-network interface that supports dynamic modification of the network's behavior. Such dynamic control is potentially useful on multiple levels:

- For a network provider, active networks have the potential to reduce the time required to deploy new protocols and network services.
- At a finer level of granularity, active networks might enable users or third parties to create and tailor services to particular applications and current network conditions.
- For researchers, a dynamically-programmable network offers a platform for experimenting with new network services and features on a realistic scale without disrupting regular network service.

The *programming interface* supported by the active network defines the "virtual machine" present at each node of the network. Depending on its design, such a virtual machine can provide a generic mechanism such as a language interpreter at each node or it may just allow the user of the network to choose a service from a set of services provided at each node. Particular programming interfaces for active networks are implemented by node-resident programs called *Execution Environments* (EEs).

This paper describes the design and implementation of the Bowman node operating system, a software platform for implementing execution environments within active nodes. Bowman was specifically implemented as a platform for the CANEs EE [1]; however, it provides a general platform over which other EEs may also be implemented.

The design goals for Bowman are the following:

- **Support per-flow processing.** Bowman provides system support for long-lived flows. Flows are classified using an efficient and flexible packet classification algorithm. Computation on behalf of a flow occurs in its own set of compute threads. Thus, the internal packet forwarding path of Bowman is inherently multi-threaded.
- **Provide a** ***fast-path*****.** 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 overheads of multi-threaded processing.
- **Enable a network-wide architecture.** 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.
- **Maintain reasonable performance.** Bowman is multi-processor capable and provides deterministic performance in user-space by utilizing POSIX real-time extensions for processor scheduling [2]. Even though Bowman is implemented entirely in user space, it delivers high performance. IP forwarding through Bowman saturates 100 Mbps Ethernet, and the Bowman classifier is able to classify packets at gigabit rates while matching on multiple fields.

The Bowman software architecture is highly modular and extensible. All the major parts of a Bowman node —communication protocol implementations, routing protocols and associated data structures, code-fetch mechanisms, per-flow processing and output queuing mechanisms— are loaded dynamically at run-time. The Bowman implementation can be ported to UNIX systems that support the POSIX system call interface [2].

In the next section, we provide an overview of Bowman. We start with the architectural context for the development; we then discuss the abstractions supported by Bowman and provide an overview of the network architecture. In Section III, we present selected details of the Bowman implementation, with a focus on the packet processing path and the packet classifier. We present a set of performance results for Bowman in Section IV including details of the forwarding performance and individual overheads in the packet processing path. We also provide a performance analysis of the Bowman packet classification algorithm and the effect of real-time processor scheduling. In Section V, we describe how the CANEs EE is implemented over Bowman. We present a survey of related work in Section VI, and conclude in Section VII.

Work supported by DARPA under contract N66001-97-C-8512.

II. Overview

In this section we provide an overview of the design and implementation of Bowman.

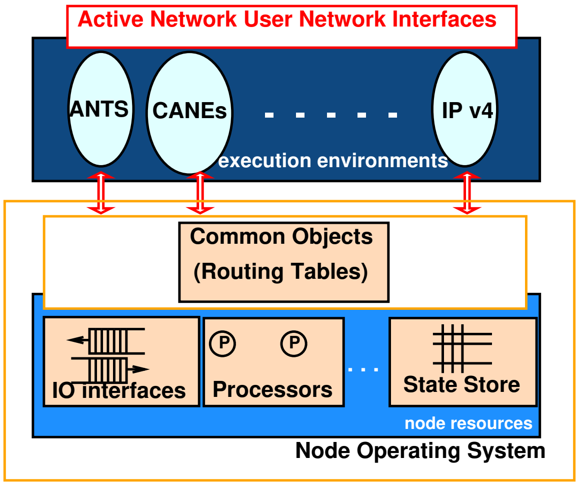

Fig. 1. DARPA active networking architecture

A. Architectural Context

Figure 1 shows the architecture for active network nodes developed within the DARPA active network research community [3]. The primary functional components in the architecture are the *Node Operating System* (NodeOS) and the *Execution Environments* (EEs). Generally speaking, the NodeOS is responsible for managing the local resources at a node, while each EE is responsible for implementing a User-Network interface, i.e., a service exported to users of the active network. In particular, each EE defines a "virtual machine" that interprets the packets delivered to it at each node. Thus an Internet Protocol implementation might be considered a very simple EE, whose virtual machine is "programmable" only to the extent of controlling where packets are delivered. On the other hand, the ANTS execution environment [4] uses a virtual machine that interprets Java bytecodes.

To understand how Bowman fits into the picture, it may be useful to consider the purpose of the programming interface exported by each component, along with its intended "users":

- The NodeOS exports an API that provides access to node resources including computing, storage, and transmission bandwidth; its "users" are the Execution Environments, or more precisely the programmers who implement them. A functional specification of this API is currently in draft [5].
- The Execution Environment provides basic end-to-end services and some form of programmability—i.e., some way of composing or extending basic services to create new ones. A primary goal of the node architecture is to enable the network as a whole to provide multiple user-network interfaces; thus the primary reason for the NodeOS's existence as a separate entity is to support multiple EEs, which may offer different forms and degrees of programmability. The basic abstractions and form of composition depend on the EE. The EE's users are the end-system *active applications* whose packets invoke and/or customize the EE's basic services (again, the real users of the EE API are the programmers who create the active applications).
- The active applications provide an interface by which end-users invoke (by sending packets to an EE with appropriate instructions) and possibly customize (via in-band or out-of-band signaling) their services.

The ability of EEs to provide interesting and novel services depends greatly on the degree to which the NodeOS exposes lower-level mechanisms. For example, providing active applications with fine-grained control over scheduling requires a greater degree of access to output and processor scheduling decisions than is traditionally provided in general-purpose operating systems. In designing Bowman we have attempted to provide this kind of access without re-implementing available components. Thus Bowman makes use of an underlying *Host OS*, and attempts to hide the differences between low-level capabilities (e.g., network interfaces) across different platforms by providing a consistent interface.

While the NodeOS/EE interface generally marks the separation of local and global concerns, the NodeOS cannot totally ignore end-to-end considerations. For example, the NodeOS is responsible for seeing that packets transmitted by end users —including both data and signaling packets— reach the proper nodes and the proper EE at a node. Bowman provides a packet-matching mechanism enabling EEs to describe packets they want to receive. In addition, Bowman makes it possible to group and identify channels consistently across multiple nodes, so that EEs can define and use *virtual topologies*. In addition, the role of the NodeOS (and Bowman) includes support for *common objects* such as routing tables that are likely to be useful to all EEs.

Bowman implements a subset of the emerging DARPA Node OS interface [5]. The interface exported by Bowman is intended to be used primarily by EE-implementors, although it can be used for other purposes as well, such as experimentation with different network architectures.

B. Bowman Abstractions

Bowman is built around three key abstractions: *channels*, *a-flows*, and *state store*.

- **Channels.** Channels in Bowman are communication end-points 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 as well. Details of the Bowman channels with examples are presented in Section II-D.
- **A-Flows.** The a-flow[1] is the 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 (i.e., user of the system). Bowman provides interfaces to create, destroy and run a-flows. Further, multiple threads may be

[1] The term "flow" is often used to describe per-user computation state at an active node [5]. Since these flows are, in a manner, *active*— they refer to both a set of packets and some processing— we adopt the use of the term "a-flow".

active within a Bowman a-flow—the level of concurrency within an a-flow is user-selected.

- **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 data sharing between a-flows without sharing program variables.

C. Bowman Structure

One of the goals of our implementation is for Bowman to be widely deployable and run on a variety of hardware. To this end, it is built as an extensible library of C-language function calls; in this section we describe the parts of the implementation.

C.1 The Host OS

In order to implement the channel, a-flow, and state-store abstractions, Bowman requires interfaces to lower level hardware-specific mechanisms such as memory management, thread creation and scheduling, synchronization primitives, and I/O devices. The Bowman implementation is layered on top of a *host* operating system that provides these lower level services. The host operating system implements hardware-specific routines for device and memory access, synchronization etc.[2] Thus, Bowman provides a uniform active node OS interface over different hardware and operating systems. Bowman can be ported to operating systems that support the *POSIX* system call interface. The current Bowman implementation has been ported to SunOS (versions 5.5 and higher) and Linux (kernel versions 2.2.0 and higher).

C.2 Bowman Extensions

The goal of active networking is to provide enhanced network *services* to network users. The Bowman channel, a-flow, and state-store abstractions provide basic capabilities, but more sophisticated services may be needed. Bowman provides an *extension* mechanism that is analogous to loadable modules in traditional operating systems. Using extensions, the Bowman interface can be extended, dynamically, to provide support for additional abstractions such as queuing mechanisms, routing protocols, user-specific protocols and other network services. The extension mechanism provides a common underlying foundation for EEs that provide various composition mechanisms. Bowman extensions are written using Bowman system calls and calls exported by other extensions. In the latter case, all required extensions must already be loaded into the system.

C.3 EEs on Bowman

EEs running on Bowman can choose from at least two models of operation: the monolithic approach and the multi-a-flow approach. The code for an EE can be loaded as an extension to Bowman and typically contains control code for the EE and a set of routines that implement the user-network interface supported by the EE. In order for the EE to process packets, the control code must spawn at least one a-flow. In the monolithic approach, the EE creates exactly one a-flow and does not expose its internal processing structure to Bowman. The EE submits a blanket request for all packets "addressed" to the EE to be delivered to that a-flow, which may do its own additional demultiplexing.

In the multi-a-flow approach, the EE code spawns one control a-flow that can be used for the EE signaling and housekeeping. The control a-flow starts a different a-flow for each user's packets, and submits a request for only that user's packets to be delivered to that a-flow. Note that in each case, third-party code, loaded on behalf of users of the network, is linked against the API that is exported by the EE; however, in the multi-a-flow approach, the user's computation is scheduled by Bowman (and not by the EE itself).

D. Bowman Abstract Topologies

Along with the node architecture described above, Bowman implements a network-wide architecture at the node OS level by providing support for *abstract links* and *abstract topologies*. Abstract links provide connectivity between a set of Bowman nodes and implement protocol processing as well as data transmission. Bowman channels are end-points of abstract links. Named sets of abstract links form named *abstract topologies*.

Abstract topologies implement user-defined connectivity over arbitrary physical topologies [3]. Abstract topologies have globally unique names (assigned when the topology is created) and all packets transmitted and received by a Bowman node traverse some abstract topology. Abstract topologies in Bowman can be used to instantiate overlay networks (much like "virtual private networks" in the Internet) in which each node possesses a set of specified attributes (e.g., a particular EE is present at each node). When a packet is received at a Bowman node, it is classified to determine to which abstract topology it belongs; packets that belong to no topology are discarded. A-Flows *subscribe* to subsets of packets on specific topologies; matching packets are demultiplexed to proper a-flows. In order to send a packet to a neighboring node on a particular abstract topology, an a-flow transmits the packet on a channel that belongs to the topology and has the required node as the other end-point. By default, Bowman abstract topologies do not provide any resource guarantees at active nodes or isolation of data between topologies; however, such properties can be added as a matter of policy during topology instantiation.

We conclude this overview of Bowman with an example illustrating how the Bowman abstract topology capability can be used to realize a network architecture (Figure 2). In Figure 2, we show a network of five nodes ($a \ldots e$) and their physical connectivity. The next step in realizing an abstract topology is to create a set of abstract links between the nodes. Note that abstract links can be defined using MAC protocols such as `Ethernet` or net-

[2] Operating systems, of course, provide additional services, such as protection between programs running in different address spaces, and security mechanisms based on (weak) notions of "principal". Presently Bowman does not depend on services other than mentioned above, and would work as well on a Host OS that does not provide such additional services.

[3] Obviously, abstract topologies cannot provide connectivity that is not supported by the transitive closure of the underlying physical adjacency relation. However, by using abstract links that implement protocols such as IP, abstract topologies may provide *logical* adjacency between nodes that have more than one physical hop between them.

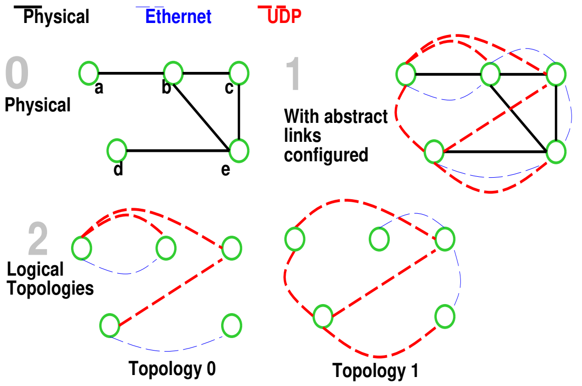

Fig. 2. The Bowman network-wide architecture: abstract links that incorporate protocol processing are overlaid on the physical topology. An abstract topology is a named collection of abstract links.

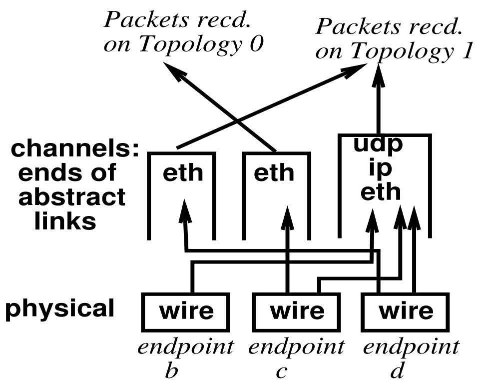

Fig. 3. Details of links and channels at node e due to the creation of abstract links and abstract topologies as shown in previous figure

work/transport protocols such as IP and UDP. Abstract links that implement protocols such as UDP are akin to UDP tunnels. In the last step, abstract topologies are created by identifying sets of abstract links. When an abstract link with a Bowman node as the end-point is created, a corresponding channel is set up at the Bowman node. Figure 3 shows the channels created at the Bowman node e due to the creation of the abstract links in Figure 2. Physically, node e is connected to nodes b, c, and d. In the example, three abstract (point-to-point) links are created with node e as the endpoint. As shown in Figure 3, each link results in one channel being created at node e. As abstract topologies are configured in the network, the channels at each node are associated with particular topologies. This information is used by routing protocols to send data to particular nodes over particular topologies.

We have designed the ALP (Abstract Link Protocol) and the ATP (Abstract Topology Protocol) protocols to create abstract links and disburse abstract topology information. These two protocols have been implemented as a-flows in Bowman.

III. Bowman: Implementation Details

In this section, we present selected details from the implementation of Bowman. Specifically, we present details of the Bowman packet processing path, packet classification algorithm, a-flow scheduling and Bowman timers.

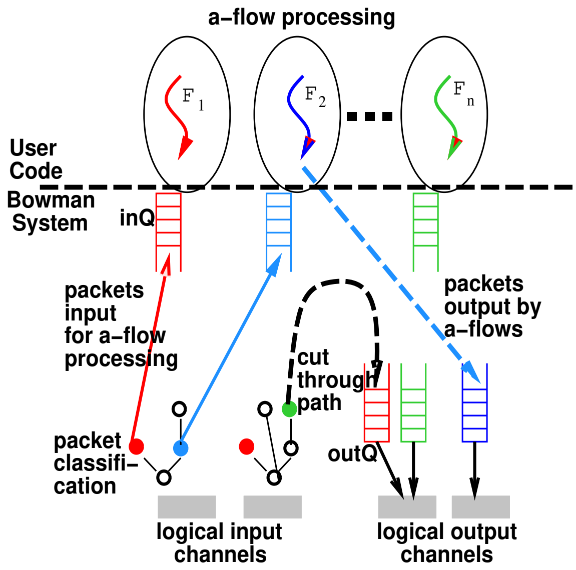

Fig. 4. Bowman packet processing path.

A. Packet Processing Path

Figure 4 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 and output scheduling algorithms, it is possible for multiple input and output threads to be active within the Bowman. The semantics and implementation of Bowman timers are detailed in the next section; in the rest of this section, we provide an overview of the input/output processing within 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 needed by the packet. Such processing is one of three types:

- **A-Flow processing.** The packet can be assigned to an a-flow for flow-specific processing. The system thread enqueues the packet into the a-flow input queue (assuming the input queue is not full—otherwise the packet is discarded). The a-flow processing thread eventually dequeues the packet and processes it.
- **Cut-through forwarding.** As shown in Figure 4, packets may be enqueued directly on an output queue, without undergoing any a-flow processing. This processing corresponds to *cut-through* paths through the Bowman node.
- **Channel input.** Specified packets received on a channel can be routed as input to other channels. For example, packets received on an IP channel with the IP protocol number

equal to 17 (UDP) may be input to an UDP channel. In this manner, Bowman supports hierarchical protocol graphs similar to the x-kernel [6]. Bowman protocol graphs are run-time configurable.

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). A-flows can transmit packets on particular topologies: the tuple (topology, routing metric, destination) identifies a channel on which the packet must be sent.

The output queuing mechanism employed within Bowman is dynamically loaded at boot time. This provides the flexibility to configure a Bowman node with a wide range of output queuing and scheduling algorithms. Our Bowman implementation provides the requisite primitives (such as system threads and fine-grained timers) for the output schedulers to support different services such as per-flow queues, fair sharing of output bandwidth, etc.

B. Bowman packet classifier

Efficient packet classification to identify flows is an essential part of flow-specific processing. The complexity of packet classification algorithms([7], [8], [9]) 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.

As a part of the Bowman, we have implemented a packet classifier that can match on an arbitrary number of fields in the packet header. 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 classifier implements a trie-based search algorithm. The nodes in the trie are constructed from a set of packet filter rules; each node in the trie corresponds to a field to be matched on the packet header. Each node in the classification trie contains an unique field match — packet filters with common prefixes share nodes in the classification trie. Within Bowman, a packet classifier is allocated for each channel. Subscriptions to the channel are specified as packet filter rules and are stored as up-calls inside the trie nodes that terminate the filter rule. When a packet arrives on the channel, the classifier trie is traversed. It is possible for incoming packets to match more than one filter in the trie; the values returned by the packet classifier depend on how it is configured (return first, best or all matches).

The Bowman packet classification algorithm is not dependent on matching particular protocols, instead, using a small language, protocols can be dynamically *taught* to the classifier at run-time. An example of this language that teaches a packet classifier the Ethernet header is given below.

```
(protocol ethernet (hdr_len 14)
    (field dst offset 0 length 6)
    (field src offset 6 length 6)
    (field proto offset 12 length 2))
```

The packet filter rules themselves are specified in ASCII text and specify a set of headers and fields to be matched. The following example shows a filter expression that matches IP packets over an ethernet link; further, the IP datagram must be from a source with DNS name `Discovery.Space.Net` and be destined to `Earth.Space.Net`. Further, the IP protocol must be UDP, and the UDP destination port number must be 2001: `[ethernet (proto = ETHERTYPE_IP) | ip (src = Discovery.Space.Net)(dst = Earth.Space.Net) (proto = IPPROTO_UDP) | udp (dport = 2001)| *]`.

The Bowman classifiers export calls to create and destroy classifiers, teach new protocols to an existing classifier, add filter rules to a classifier, and to match a buffer against a set of rules.

C. A-Flow scheduling and timers

The Bowman packet processing path incorporates multiple threads and queues; thus, the Bowman implementation has to context-switch between threads in order to process each packet. Further, asynchrony due to the queues on the packet processing path introduces delays for each packet. Two specific features of Bowman help in minimizing these overheads. 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.[4] As we will see in the performance figures in the next section, 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.

Bowman 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.

IV. Bowman Performance

In this section, we present a set of Bowman performance results, beginning with base forwarding performance. Note that this base performance represents an upper bound on the throughput achievable by a single flow through a Bowman node; actual throughputs for user flows will certainly depend on the flow-specific processing at each node. Thus, the goal of this section is to quantify the overheads incurred due to Bowman itself. To that end, in Section IV-A.1, we analyze the performance of each component of the packet processing path. In Section IV-B, we provide results of the performance of the Bowman packet classifier. Because the classifier can handle packets faster than the test network can transmit packets, we present results that are in-

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

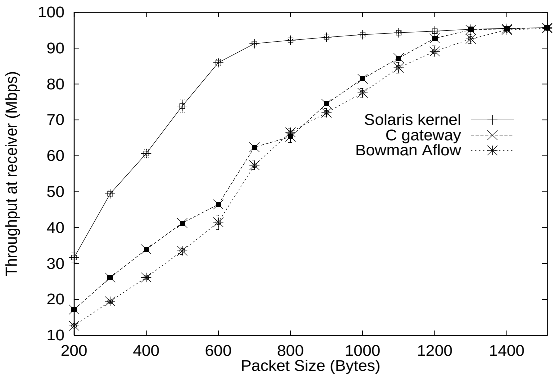

Fig. 5. Sustained throughput

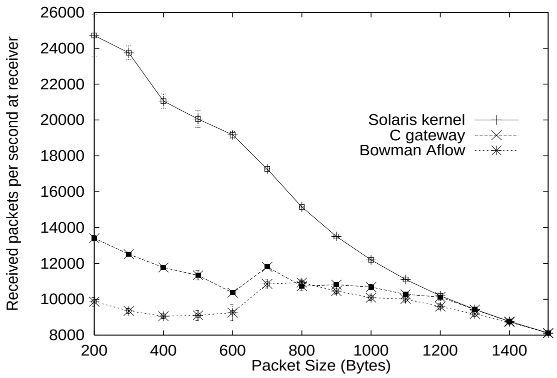

Fig. 6. Sustained packet processing rate

dependent of the network, and depend only on the test machine processor.

We conclude the section with an analysis of the effects of real-time scheduling for a-flows.

A. Forwarding performance

The forwarding performance experiments were performed on a three-node testbed configured in a straight line topology. The interior node executed Bowman. The edge nodes were Sun Ultra-5 300 MHz machines connected via 100 Mbps Ethernet to the Bowman node. The Bowman node was a Sun Ultra-2 2-processor machine with each processor rated at 168 MHz. The edge nodes ran the SunOS 5.7 operating system while the interior node executed SunOS 5.5.1. 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. For this experiment, both the read and the write socket buffer sizes for the C gateway were set to 256 Kbytes. 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). All the times reported were gathered using the Solaris high (nanosecond) resolution timer[5].

Figure 5 shows the sustained lossless throughput through Bowman, the C gateway, and the Solaris kernel as measured at the receiver. Each data point is an average of 30 samples where each sample measured the time to receive 1000 packets of the given size. Figure 6 shows the raw packet rate (in packets/second) at the receiver for the same experiment.

Compared to kernel forwarding, Bowman incurs four extra overheads: (1) overhead for at least two context switches for data read and write, (2) overhead for data copies in and out of user space, (3) overhead for dispatching system calls, and (4) overheads due to the queues in the Bowman packet processing path. The C gateway incurs the first three overheads as well[6].

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. We should note that the Bowman performance results shown here are for a path that includes an a-flow; Bowman cut-through processing results are similar to the C gateway results [7].

A.1 Overheads on the Packet Processing Path

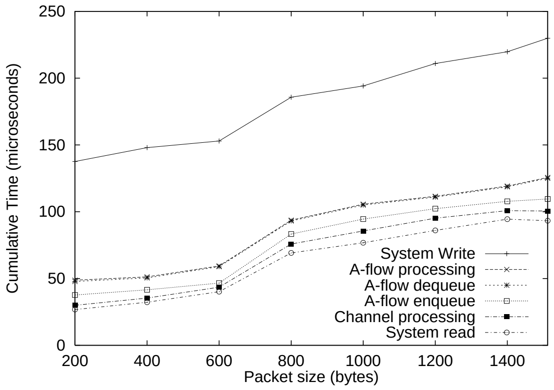

Fig. 7. Overhead of Bowman components

In this section, we quantify the overhead of Bowman multi-threaded processing, context switches and internal queues. In

[5]Please see `gethrtime(3c)` manual page on SunOS 5.5+.

[6]Note that, in general, Bowman multiplexes multiple input/output channels and may incur an extra overhead due to a `poll(2)` system call (called by `select(3c)`) not present in the read-write loop for the C gateway. In our implementation, Bowman is optimized not to call `poll` unless there is no data available on its last input descriptor.

[7]All the user-space results (Bowman and the C gateway) presented here are for a 32-bit executables; we are in the process of porting Bowman for a native 64-bit platform. At the very least, this will alleviate some of the system copy overheads seen in these results.

order to quantify this overhead, we augmented the Bowman packet processing path with several high resolution timers that allowed us to record precise measurements of the time consumed by each component of the Bowman packet processing path. Augmenting the packet processing path with these timers degraded the performance of the system by about 5%.

Figure 7 shows the *cumulative* processing time taken by each step in the Bowman packet processing path. The Bowman processing is quantified in the following steps: (1) channel processing: this includes allocation for packet memory from a dedicated packet ring, a trivial packet classification (the packet classifier is called with a filter that matches all packets) and the subsequent demultiplex that calls the a-flow enqueue function; (2) the a-flow enqueue; (3) the a-flow dequeue (by the a-flow processing thread); (4) null a-flow processing; and (5) the Bowman channel dispatch on write and deallocation of packet memory. Since Bowman does not internally copy packets, the Bowman overhead is relatively constant at about 25 μ-seconds for all packet sizes. Of this, about 5 μ-seconds are incurred due to the timers. The processing time is, in fact, dominated by the system read and write calls. For small packet sizes (200 bytes), the system read and write calls take approximately 24 μ-seconds and 90 μ-seconds, respectively, and the overhead due to Bowman is around 15%. However, for large packet sizes (1514 bytes), the system read and write calls take 95 and 125 μ-seconds respectively and the Bowman overhead reduces to approximately 10%.

B. Packet classifier performance

Number of Filter Rules	Avg. Classification Time (nanoseconds)
1	2578
2	2853
4	3325
8	3628
16	3909
32	5192
64	6806
128	6720
256	7477
512	8187
1024	9952

TABLE I
VARIATION OF AVERAGE PACKET CLASSIFICATION TIME VS. NUMBER OF FILTER RULES

In this section, we present the results of two experiments that analyze the performance of the Bowman packet classifier algorithm. In the first experiment, we report the packet classification time as the number of rules in the classifier is varied. These results were generated on a 300 MHz Ultra-5 SunOS 5.7 machine with 32-bit executables. The filter rules were generated for a packet with Ethernet, IP, UDP, and an application-level header. All four protocols (including the application-specific protocol) were taught to the packet classifier. The filter rules were created using a program that generates fields and values for the fields uniformly at random using the 48-bit UNIX pseudo-random number generator[8]. On average, each filter rule had approximately six fields and the classifier was configured to find all the matches. The average classification time as the number of rules were varied is shown in Table I. Note that in Bowman, packets are classified on the ends of channels — as such we do not expect a large number of rules to be added to a classifier (as each rule can result in an a-flow at the node). Nonetheless, the Bowman packet classifier performs well with a relatively large number of rules, e.g., with 256 rules, each match takes about 7.4 μ-seconds. This corresponds to a classification rate of over 1.3×10^5 packets per second (or a throughput of about 1.5 Gbps assuming 1514 byte packets).

Number of Fields	Avg. Classification Time (nanoseconds)
1	3174
2	3809
3	4095
4	4788
5	5117
6	5417
7	5687
8	6189
9	6486
10	6730
11	7017

TABLE II
CHANGE IN CLASSIFICATION TIME VS. NUMBER OF FIELDS IN THE BEST MATCHING FILTER RULE.

Table II shows the variation in packet classification time as the number of fields in the best matching rule is increased. In each case, the classification trie consists of exactly 32 filter rules; however, the number of fields in the best matching filter rule is varied. Further, the experiment with k fields in the best matching filter rule includes the previous $(k-1)$ best matching filter rules in its set of 32 filter rules, e.g., if the best matching filter is `[ip (src = alpha) (dst = beta) | *]` then the rule `[ip (src = alpha) | *]` is also included in the rule set. We report the results for the case when the classifier finds all matching rules in the rule set; thus, in case the best matching rule has k fields, the classifier outputs k matching rules. In Table II, we see that there is an approximately linear increase in the classification time as the number of fields is increased. This is because common fields share the same node in the classification trie and each new field adds a new node to the trie. The extra time in consecutive measurements is due to the traversal of this new node (and the corresponding time it takes to access the new field in the matched packet's header).

C. Effect of real-time scheduling

In Figure 8, we present a snapshot of the queue lengths in a running Bowman system to show the effect of real-time scheduling.

[8]Please see the `drand48(3c)` manual page.

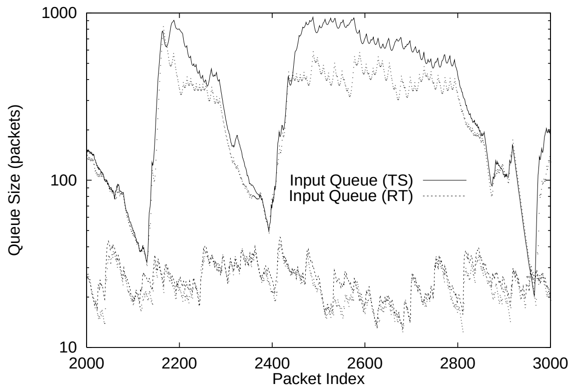

Fig. 8. Effect of real-time scheduling on system and a-flow queue sizes. The dots under each curve correspond to the sizes of the output queues.

The two curves in the figure correspond to a-flow input queue sizes when the system scheduler is run in time-sharing mode (TS) versus real-time (RT) mode. Under each curve is a set of dots that show the size of the output queues for the corresponding time period. There are three different kernel threads active: the Bowman input thread enqueues packets into the a-flow input queue. The a-flow input queue is drained by the a-flow thread which then enqueues the packet in the a-flow output queue; the a-flow output queue is drained by the Bowman output thread. Under time-shared scheduling, these threads are scheduled on far too coarse a quantum; as such, the queue sizes fluctuate wildly and the average queue size is over 300 packets. (Note that larger and widely varying queue sizes also imply longer delays and more jitter through the node.) In the real-time case, we run each kernel thread with the same real-time priority and give each thread a time quantum of 1000 nanoseconds. Due to the deterministic and fine grained scheduling, the queue sizes are smaller by an order of magnitude and the system is far more stable. For this experiment, the priority of each thread was static; in subsequent work, we will investigate dynamic processor priority scheduling schemes for Bowman threads.

V. Configuration of Bowman for Active Networking

In this section, we give an overview of how Bowman is being used for active networking. Specifically, we discuss:

- loading of the EE and user code into Bowman as extensions;
- structuring the EE and user computations across multiple a-flows;
- using extensions for accessing node-wide state such as routing tables; and
- the set of support protocols and extensions required to configure a working active node.

CANEs over Bowman

The CANEs[1] execution environment allows network programmability on a per-flow basis and has been implemented using the system call interface exported by Bowman. The CANEs EE is loaded as an extension to Bowman. Initially, the EE creates a signaling a-flow; the signaling a-flow creates a signaling channel on a well-known UDP port and subscribes to all messages that arrive on that channel. A CANEs signaling message consists of three parts: a computation part that specifies a set of code modules required by the flow, a data part that is fed as input to the computation, and an input/output part that specifies the set of packets to be input to the computation. When the signaling a-flow receives a valid user request, it downloads the requested code modules from a code server; the code server(s) are specified as part of the signaling message. The protocol to fetch code is loaded as an extension into Bowman and is a generic service provide by the node to the EE. The code fetch protocol exports a well-known interface that is used by the CANEs EE to download the code required by per-flow computations. The downloaded code modules for CANEs are in the form of shared objects that are linked into the Bowman node using the Bowman extension mechanism. In case of flow-specific code, however, the newly linked symbols are not exported (as is the case for regular Bowman extensions) as system calls; instead, the CANEs signaling thread extracts specific functions from the linked-in shared object using names supplied as part of the signaling message. When all the code modules required for a flow are retrieved and linked into the Bowman node, the signaling thread spawns a new a-flow. The new a-flow subscribes to the packets specified in the signaling message (possibly creating new channels in the process) and then executes the flow-specific computation.

Routing Extensions

Typically, flow-specific computations do not implement their own routing protocols; instead a set of routing protocols are linked into the node as extensions and execute within their own a-flows. These routing protocols use the Bowman state-store mechanism to export routing tables for different abstract topologies (and different routing metrics). The routing protocols also export well-known methods to access these routing tables. Unless an user flow implements a routing protocol, it uses the generic routing services exported by the routing protocols to forward its data.

Support Protocols

Along with the code fetch and routing protocols, a number of other extensions are also routinely initialized as part of a Bowman node. These include extensions for output queue scheduling; specific protocol implementations such as Ethernet, IP, UDP, and TCP that are required to implement channels of specific types; protocols that provide name service such as DNS and ARP; multi-threaded timers that can be used by per-flow computations; protocols for creating and managing abstract links and topologies. Lastly, a login shell for Bowman called *Hal*, is also loaded as an extension: *Hal* provides an extensible command interpreter that can be used to query the state of various objects —such as network interfaces, channels, and a-flows— that are present in a Bowman node.

VI. Related Work

In this section, we provide a survey of work related to Bowman: this includes the areas of system support for flow-specific processing at routers and development of new operating systems.

A software architecture for flow-specific processing has been developed as part of the Router Plugins project [10] at Washington University. This work has adapted a NetBSD kernel to incorporate user-processing at specific points (*gates*) on the IP forwarding path with about 8% overhead. The major feature of Bowman that is missing from Router Plugins is the general set of communication abstractions —channels, abstract links and topologies— that is supported by Bowman; this reflects the IP-based focus of the Router Plugins project. Router plugins classify incoming packets to specific flows using a packet classification scheme similar to Bowman: the packet classification step identifies the function that is bound to a packet at each processing gate. As the packet processing path in Router Plugins is integrated with the flow-specific processing, Router Plugins do not incur the thread-switching and queuing overheads of the multi-threaded multi-queue packet processing path of Bowman. Architecturally however, Router Plugins merge Node OS and EE functionality and thus, unlike Bowman, Router Plugins do not export a node OS interface over which different EEs can be built[9]. This architectural difference between Bowman and Router Plugins clearly exposes the performance versus generality tradeoff inherent in the design of extensible router platforms.

The aim of Practical Active Network (PAN) [12] project at MIT is to build a high performance *capsule*-based active node that is based on the ANTS [4] framework. Experiments have shown that the forwarding performance of capsules containing Java [13] byte-code is dominated by overhead incurred due to execution within the Java virtual machine. PAN in-kernel performance is comparable to Bowman user-space performance; in their experiments with 1500 byte capsules containing native code, executed in-kernel, they were able to saturate a 100 Mbps Ethernet. Unlike PAN, Bowman does not mandate that code be transported in band; instead the code transport and execution mechanism is implemented by EEs that execute over Bowman. Architecturally, PAN is an in-kernel implementation of an EE using Linux as the node OS.

Another effort based on ANTS is the development of a new operating system called Janos [14] at the University of Utah. Janos implements a special Java Virtual Machine and Java runtime for executing Java byte code. Janos includes a modified version of ANTS that supports resource management. Their main design goal is to provide a strong protection and separation mechanism between different user code executed at the active node. While execution of user code within a virtual machine (like Java) provides a high degree safety of execution, it also has associated performance costs. Unlike Janos, Bowman does not enforce safety constraints on loaded code at run time. During the signaling phase in Bowman, the code-fetching mechanism must decide, based on credentials provided by the user and the node security policy, whether it trusts the requested code; if not, the request is denied. This is the time-honored approach of relying on the code supplier for safety in order to obtain better performance. The choice of "C" programming language to implement the Bowman system call interface has given us good performance, but probably sacrifices some flexibility (code can only come from *a priori*-trusted parties) compared to Java-based approaches.

With an aim to push functionality from end devices into the network, the Extensible Routers project [15] at Princeton University has designed an open, general purpose computing/communications system. They share the same design goals as Bowman: support complex forwarding logic and high performance while reusing available building blocks. The architecture for Extensible Routers is based on ideas of I/O paths from the Scout [16] operating system and I/O optimizations developed for the SHRIMP [17] multi-computer.

[9] An extension to Router Plugins for active networking, "Active Router Plugins" [11] has been designed.

VII. Conclusions

We have designed and implemented Bowman, a node OS for active networks. The software architecture of Bowman is highly modular and dynamically configurable at run-time. The Bowman design and implementation is decoupled from any particular network protocol or routing scheme. Bowman provides system-level support for abstract topologies that can be used to create virtual networks. The Bowman multi-threaded packet processing path is an example of how to structure system software for routers that may support significant per-flow processing. Further, we have shown how real-time processor scheduling can be used to provide deterministic packet forwarding performance even in user-space. Our user-space implementation is portable to systems that support the POSIX system call interface, though of course specific Bowman capabilities that use capabilities of the underlying system beyond POSIX may not be available on all platforms.

Our results show that Bowman packet forwarding performance is relatively high: Bowman is able to saturate 100 Mbps ethernet with only 1400 byte packets. The Bowman packet classifier implements a protocol-independent algorithm that can classify packets at gigabit rates in software.

Our work on Bowman has been motivated by the need for a node OS for the CANEs EE. In future, we hope to port other EEs to execute over Bowman. Bowman provides a flexible platform for experimentation with a number of major open issues in active networking, including algorithms for integrated processor and link scheduling; algorithms for allocation of resources to flows within active nodes; and policies for instantiation and isolation of different abstract topologies. We hope to use Bowman as a research vehicle to investigate some of these open problems.

VIII. Acknowledgments

We are grateful to Matt Sanders for helping us at various stages of software development. He has also provided us extensive and insightful comments on a draft of this paper. We would also like to thank Youngsu Chae for his contributions on the Bowman abstract topologies and their implementation. This work is supported by Defense Advanced Research Projects Agency under contract N66001-97-C-8512. Information about the Bowman software can be found at: <http://www.cc.gatech.edu/projects/canes/software.html>

References

[1] K. Calvert, S. Bhattacharjee, E. Zegura, and J. Sterbenz, "Directions in active networks," IEEE Communications Magazine, 1998.

[2] *System Application Program Interface (API) [C Language]*, Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

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

[4] 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.

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

[6] Norman C. Hutchinson and Larry L. Peterson, "Design of the *x*-kernel," in *Proceedings of the SIGCOMM '88 Symposium*, Stanford, California, 16–19 August 1988, pp. 65–75, ACM Press.

[7] Mary L. Bailey, Burra Gopal, Micheal A. Pagels, and Larry L. Peterson, "PATHFINDER: A pattern-based packet classifier.," in *Proceedings of the first USENIX Symposium on Operating Systems Design and Implementation*, Nov 1994.

[8] V.Srinivasan, S.Suri, and G.Varghese, "Packet Classification using Tuple Space Search," in *Proceedings of ACM SIGCOMM*, Sept 1999.

[9] Pankaj Gupta and Nick McKeown, "Packet Classification on Multiple Fields," in *Proceedings of ACM SIGCOMM*, Sept 1999.

[10] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner, "Router Plugins: A Software Architecture for Next Generation Routers," in *Proceedings of SIGCOMM '98*, Vancouver, CA, Sept 1998.

[11] Dan Decasper, *A Software Architecture for Next Generation Routers*, Ph.D. thesis, Washington University, 1999.

[12] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek, "PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems," in *Proceedings of IEEE OpenArch '99*, March 1999, pp. 78–89.

[13] Ken Arnold and James Gosling, *The Java Programming Language*, Addison Wesley, Reading, Massachusetts, 1996.

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

[15] Larry Peterson, Scott Karlin, and Kai Li, "OS Support for General-Purpose Routers," in *HotOS Workshop*, March 1999.

[16] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman, "Scout: A communications-oriented operationg system," Tech. Rep. 94-20, Department of Computer Science, The University of Arizona, 1994.

[17] "SHRIMP: Scalable High-performance Really Inexpensive Multi-Processor," http://www.cs.princeton.edu/shrimp/.