PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems

Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek
MIT Laboratory for Computer Science
Cambridge, Massachusetts

Abstract—**A capsule-based *active network* transports *capsules* containing code to be executed on network nodes through which they pass. Active networks facilitate the deployment of new protocols, which can be used without any changes to the underlying network infrastructure. This paper describes the design, implementation, and evaluation of a high-performance active network node which supports multiple mobile code systems. Experiments, using capsules executing unsafe native Intel ix86 object code, indicate that active networks may be able to provide significant flexibility relative to traditional networks with only a small performance overhead (as little as 13% for 1500 byte packets). However, capsules executing JavaVM code performed far worse (with over three times the performance overhead of native code for 128 byte packets), indicating that mobile code system performance is critical to overall node performance.**

Keywords—**active networks, networking protocols, PAN, ANTS, performance, mobile code, Java, capsules, software segments.**

I. Introduction

PAN IS A NEW ACTIVE NETWORK SYSTEM that provides a foundation for building practical active networks. PAN's eventual goal is to provide performance comparable to existing passive networks while providing a safe execution environment for mobile code. Experience with PAN indicates that the high degree of flexibility provided by capsule-based active networks can be obtained with little performance overhead.

By allowing application-specific computation within the network, *active networks* [1] enable new protocols to be deployed quickly, without the need for a lengthy standardization process or for the installation of new hardware. Rather than standardizing protocols that describe how nodes should forward packets, an active network standardizes an execution environment for *capsules* that pass through network nodes. A capsule contains both data and a reference to code to execute when it passes through a node. In a traditional network, routers examine packet headers and decide where to forward the packets. In a capsule-based active network, routers execute the code referenced by capsules, and this code tells the router where to forward the capsules.

The implementation of a practical active network poses several challenges. Its performance must be comparable to that of existing networks. It must be at least as safe, secure, and robust as existing networks — a difficult challenge when code can migrate and execute within a huge distributed system encompassing many administrative domains. Finally, due to the highly heterogeneous nature of an internetwork, it must have the same high degree of interoperability as a traditional network.

Because the uses for active networks are still being determined, the degree to which performance, safety, and interoperability must be addressed is still unclear. The applications that run on top of the system will have a substantial effect on all three of these requirements. For example, the requirements on performance are substantially different for a network management system than for a system that handles all of the data traffic within a network.

PAN is an active network system that takes a first step towards realizing a "practical" capsule-based active network. It is designed to run mobile code in the network node with performance comparable to existing passive networks. Although full safety is not yet provided, it should be possible to obtain this through an appropriate safe mobile code system. PAN is able to obtain performance despite executing mobile code for every capsule that passes through the node. PAN achieves performance in four ways: by processing capsules in the kernel, by minimizing data copies, by caching code for immediate execution, and by providing capsules with a flexible execution environment.

By processing capsules in the kernel, rather than in a user process, a PAN node avoids copying data between kernel-space and user-space, and it greatly reduces interference from the scheduler. By providing a uniform memory management system, which allows handles to regions of memory to be passed around the system, PAN eliminates the need for a node to copy or even touch the data body of capsules that are simply being forwarded. By caching native executable or JIT capsule code, a PAN node only incurs the cost of loading and/or compilation the first time a new type of capsule is used. By providing a flexible execution environment, PAN enables capsules to do just what they need to do, to do it in the most appropriate way, and to work with abstractions rather than fight against them.

PAN provides an encouraging answer to a fundamental performance question: in the base case, in which packets are simply forwarded to their destination, can an active network node achieve performance comparable (in bandwidth and latency) to a traditional network? Tests conducted on a 200MHz Intel PentiumPro running Linux show that when using native object code (which does not provide safety, security, or interoperability), PAN can forward at least 100Mbps of data; furthermore, compared to a traditional router running on the same Linux machine, PAN incurs a fixed overhead of roughly only 20 microseconds per capsule (13% for 1500 byte packets).

Further tests conducted on the same machine have demonstrated that when using Java bytecodes (which provide interoperability and some safety), capsule processing times increase substantially. The active overhead for 128 byte packets being processed by the JavaVM in user-space is over three times the

Research supported in part by DARPA contract No. N66001-96-C-8522, monitored by the Office of Naval Research. Contact address: nygren@mit.edu. For more information, see http://www.pdos.lcs.mit.edu/~nygren/pan/

active overhead for capsules executing native code in user-space. This indicates that high-performance mobile code systems will be crucial to high-performance active network nodes that provide safety, security, and interoperability.

The flexibility and performance of PAN make it a powerful tool for experimenting with active network technology. The architecture of a PAN node is relatively simple, flexible, and portable — the current PAN implementation supports multiple node operating system environments (within a Linux kernel and as a UNIX user-space process) and provides support for multiple mobile code systems. Almost no changes are required in the majority of the shared code base when adding support for other operating system environments or mobile code techniques. A public release of PAN will be made available later this year.

Section II summarizes other work related to this paper, Section III describes the design and implementation of PAN, Section IV presents performance results, and Section V presents our conclusions.

II. Related Work

Interest in active networks is fairly recent, having started in 1996 [1]. Surveys of active network research [2] distinguish two approaches to active networking: the *capsule-based* (or *integrated*) approach, used by PAN, and the *programmable switch* (or *discrete*) approach. The integrated approach uses the *messenger paradigm* [3]: capsules containing both code and data move through the network and are executed on the nodes they encounter. The discrete approach adds functionality to nodes out-of-band from the packets being processed by the node.

Several projects at MIT have been exploring the construction and use of active networks. The ACTIVE IP Option [4] embeds small Tcl programs in the option fields of IPv4 packets. The ANTS system [5] supports rapid prototyping and application-specific in-network processing through a capsule-based system written in Java. Several sample applications have been developed using ANTS [5], [6], [7]. The development of PAN drew heavily on many of the ideas developed in the work on ANTS. Unlike ANTS, however, PAN is designed for performance so that it can be used for real applications. PAN introduces the ideas of software segments and panStreams, has a more flexible system for assembling protocols from collections of code objects, has fewer required elements in the header of capsules, does not have a global soft state cache, and provides a simpler and more flexible interface for allowing capsules to deliver data to applications. PAN also supports multiple mobile code systems and optionally runs inside operating system kernels.

The BBN Smart Packets project [8] is exploring lightweight, but still expressive, capsule-based active networks. Many aspects of Smart Packets are similar to PAN. However, the focus of BBN's work is on network management and diagnosis, which has stronger security requirements and weaker performance requirements than PAN.

Georgia Tech is both developing a programmable switch [9] and investigating the use of active networks to reduce network congestion. The University of Pennsylvania and Bell Communications are developing a programmable switch that uses Caml and SML-based programming language technologies [10]. The University of Pennsylvania has also developed PLAN [11], a programming language for writing safe and resource-bounded capsule code. PLAN acts as a scripting language by allowing lightweight programs to use the functionality of more powerful services.

The PRONTO [12] system from AT&T Research is a discrete-approach active networking system that uses control-on-demand for achieving performance. Control-on-demand provides for different degrees of programmability, ranging from allowing packets to flow through the node without interference to performing active processing on all packets. However, PAN demonstrates that it may not be necessary to make this tradeoff for packet-oriented networks if the overhead of executing active code for every capsule is low enough.

Joust [13], developed at the University of Arizona, is a communication-oriented platform for building "Liquid Software" [14] systems such as active networks. Built on top of the Scout [15] operating system, Joust provides a high-performance Java runtime system. Experiments [13] have shown that ANTS runs substantially faster under Joust than under the combination of Linux and Sun's JavaVM.

In addition to the Liquid Software project, CMU's Proof-Carrying Code [16] and Sun Microsystems' HotSpot [17] technology both aim to develop high-performance mobile code systems. As this paper demonstrates, these technologies will be essential to high-performance active network nodes.

III. Design and Implementation

This section describes the design of PAN and provides information about its current implementation; full details can be found in Nygren's Master's thesis [18].

A. PAN architecture

PAN uses an active network architecture similar to that of ANTS [5]. In PAN, *nodes* are connected by unreliable *network links*. Any node can communicate potentially with any other node (except in the case of network failures) by sending a *capsule* across one or more network links. As capsules pass through the network, they may be processed by any or all of the nodes they encounter. Leaf nodes of the network can be connected to *applications* by *application links*.

Abstractly, a capsule contains both the data it transports and a reference to a *code object*, which contains instructions to be evaluated at each node through which the capsule passes. These instructions can direct the node to pass the capsule towards a destination node, modify the contents of the capsule, pass the capsule to an application, or access state within the node. If a code object is not yet available in a node, *demand loading* retrieves it from some other node. A *protocol* is a group of code objects that are designed to work together.

Nodes also maintain state that can be accessed by the capsules they process. In non-leaf nodes, this state is always *soft state*, so that capsules cannot rely on the persistence of data stored within the network. Nodes may be restarted periodically, the network topology may be rearranged dynamically, and old data stored within nodes may be flushed when capsules attempt to add new data.

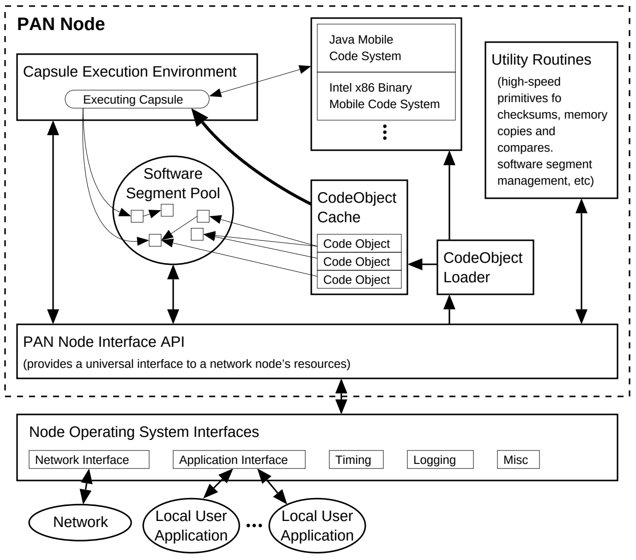

Fig. 1. The architecture of a PAN node.

B. Overview of a PAN node

A PAN node evaluates capsule code to completion in an *environment* that provides access to node resources through the PAN Node Interface (PNI). Figure 1 illustrates the node architecture, which provides a simple and consistent interface that enables code objects to run on PAN nodes on a wide range of platforms. Uniform interfaces provide portability and code mobility across different node operating systems, and they allow multiple mobile code systems to operate simultaneously within a PAN node.

Memory within a PAN node is managed through a *software segment* interface, which provides a consistent interface to code objects, minimizes data copies, allows data to be shared among code objects, and provides buffer-management functionality similar to that found in many modern operating systems. The memory management abstraction also allows multiple mobile code systems to coexist and share the bulk of the underlying node implementation.

When a capsule arrives at a node, an environment is created in which to evaluate it. The environment contains information about the origin of the capsule (so that a node can determine reliably whether the capsule was inserted by an application at the node or received over the network), the capsule's status (ready to run or waiting for a code object), and the resources the capsule is using (so that they can be freed when the capsule finishes its execution). In the current implementation, a PAN node only evaluates one capsule at a time; as in traditional networks, capsules that arrive when a capsule is being evaluated are queued for later processing.

The PAN architecture supports multiple mobile code systems, which can coexist within a node. The current PAN implementation supports both a simple (and completely insecure) system for dynamically loading Intel ix86 object code and an interface to any Java Virtual Machine [19] that supports the Java Native Interface (JNI) [20]. Java is supported currently only in user-space nodes due to the lack of an in-kernel JavaVM; work is in progress on making the Kaffe OpenVM JavaVM work inside the kernel.

C. PAN interfaces

The *PAN Node Interface* (*PNI*) presents code objects, and hence capsules, with a uniform interface to data types (e.g., hash tables) and routines for use in accessing memory and other node resources, for inserting code objects into nodes, for logging status information, for sending capsules to other nodes or applications, for determining the local node time, and for performing checksum or cryptographic operations. The Java implementation of the interface was constructed by wrapping the native implementation and providing finalization routines to free storage allocated by that implementation.

The *PAN Application Library* (*PAL*) enables applications to send and receive capsules. It contains routines for connecting to nodes, inserting capsules into nodes, and receiving data from nodes. It also provides the same uniform memory creation and access routines (software segments and software segment streams) as does PNI. A *portmapper* code object and associated PAL routine support the development of port-oriented protocols.

The PNI also provides routines that constrain the interaction between capsules and applications. One method allows capsule code to obtain an unforgeable reference to the application link that injected the capsule into the network. This reference can be stored in the persistent state of a code object and used by later capsules to deliver data back to the application. Indeed, this is the only way for capsules to deliver data to applications. Thus applications can control which code objects can pass data to them, and simple capsules (and some types of server applications) can send data rapidly to an application without any need for looking up a port mapping.

D. Code objects

PAN code objects contain executable instructions and data, which can be accessed through method or function calls. Code objects specify the type of code they contain and the API the code uses (e.g., PAN or ANTS). For the Intel ix86 mobile object code system, each code object contains a single ELF or a.out object (".o") file. Nodes link code objects together and resolve symbols using a binary object code loader derived from a Metro Link module loader to be included in XFree86 4.0. This mobile code system provides no safety or security, although although limited security could be added for some applications using cryptographic signatures. For the Java mobile code system, each code object contains a collection of Java class files that are loaded into the node using an instance of a PAN ClassLoader created for that code object.

Each code object has a unique *code object name*, which consists of a cryptographic hash (e.g., SHA-1 [21] or MD5 [22]) of the code object's code. This scheme eliminates the need for a centralized code naming system, guarantees that capsules are executed using the code objects they specify (assuming that the capsule wasn't modified on the wire and that the node has not been compromised), eliminates code versioning problems, and guarantees that code objects that reference other code objects actually use (the correct version of) those code objects. In addition, capsules enter nodes with a pre-computed hash that can be used to rapidly look up their code objects in a hash table.

Each PAN node maintains a *code cache,* keyed by code object name, that contains all code objects loaded into the node. Each capsule begins with the name of the code object that should be used to process it. On arrival, a capsule's code object is looked up in the code cache; if necessary, the code object is loaded dynamically from some node on the path to a home node specified in the capsule header [5]. Then the code object's *accept* method is applied to the capsule. In addition to the *accept* method, code objects may also have *init* and *finalize* methods that, if present, are called when the code object is loaded and unloaded.

Code objects may *depend* on other code objects, allowing a code object to call functions or access variables or classes in another code object on which it depends, thereby facilitating modular programming and code reuse. Unresolved symbols or classes within a code object are resolved using the code objects on which the code object directly depends.

Each code object maintains its own *soft state* — data that capsules leave at nodes for other capsules to access later. Soft state can be used, for example, to keep track of protocol state variables, to store capsule data for retransmission, to maintain multicast routing tables, or to maintain mappings from port numbers to application links.

Because code objects can control which symbols they export (e.g., by declaring variables and functions as `extern` rather than `static` or by declaring classes and methods as `private`), it is possible for a code object to export a narrow interface to code objects that depend on it while keeping some state information private. Since soft state is maintained within code objects rather than in a global and unified soft state cache, code objects, called *guardians*, can safely maintain shared state, such as routing tables, and require other code objects to use proper abstractions to access the state.

E. Capsules

Capsules provide the means of communication between nodes in PAN. They contain the name of a code object (a cryptographic hash), the address of a node on which the code object can be found, and an arbitrary data body. It is up to applications and code objects to decide what information goes into a capsule's data body, what needs to be checksummed, and what needs to be encrypted. Such a minimal format for capsules is motivated by the end-to-end argument [23] and by extensible systems that provide minimal core functionality in order to give maximum flexibility to application developers.

The current PAN implementation prefixes capsules with a link-level header, which contains the next hop destination, the length of the capsule, and any other link-level information required by the underlying network.

F. Memory management

PAN nodes, code objects, and applications reference regions of memory through a consistent *software segment* interface, which hides platform-dependent data structures from code objects and from the platform-independent parts of the system. Software segments also provide a uniform mechanism to track resources through reference counting. Unlike schemes for software fault isolation [24], software segments allow data to be transferred and shared between different parts of the system without any need for data copies or virtual addressing tricks.

A software segment contains a default header and optional additional header fields, which may contain pointers to associated resources. The default header is five words long. It contains a pointer to a region of data (the "contents" of the software segment), the length of the data, the (half-word) type of the software segment, a (half-word) reference count, a pointer to the next software segment in a chain, and a method for finalizing the software segment and freeing any resources associated with it.

The networking layers of most modern UNIX-like operating systems provide a buffer-management system in order to utilize memory efficiently and to minimize the number of times data is copied or touched (e.g., BSD mbufs [25] and Linux sk_buffs [26]). Software segments provide a buffer management system for PAN that is able to encapsulate the buffer management systems of different operating systems, thereby allowing code objects to run efficiently without knowing anything about the underlying operating system. For example, within the Linux kernel implementation, a capsule can be received by a PAN node and sent out across the network without copying its contents. Additionally, no copies are needed when a capsule places its contents into a node's soft state for future retransmission. Hence capsules can be placed speculatively in a soft state cache with almost no performance overhead.

Software segments allow capsules to be constructed rapidly by splicing together noncontiguous regions of memory. By chaining together two or more software segments using the "next" field, code accessing the software segments can view the data regions in software segments as being a single contiguous region of memory.

PAN software segments currently can contain regions of malloced or kmalloced memory, memory mapped files, and kernel sk_buffs. They can also keep track of references to application links, overlay portions of some other software segment, and act as containers for other software segments. Different types of software segments may add fields to the default header (e.g., pointers to sk_buff headers).

Growable *software segment rings* associated with each capsule environment or code object keep track of the software segments they reference. They enable software segments to be freed at once when a capsule terminates or a code object is released from the code cache.

Because accessing the noncontiguous regions of memory encapsulated by software segments can be tedious, PAN provides *software segment streams*, also called *panStreams*, to simplify the common tasks of reading and writing sequential items to and from a chain of software segments. A panStream contains a pointer to the current software segment, a pointer into that segment's data region, and a count of the number of bytes remaining in the data region. Operations on panStreams support seeking forward in streams, copying data between streams, reading and writing arrays of bytes, performing checksum and cryptographic checksums on regions of memory, reading and writing multi-byte data types in a platform-independent fashion (performing byte-swaps and conversions as needed), and reading and writing commonly used data types, such as node addresses and code object names. A serializer stub generator generates C code for reading and writing data structures from and to panStreams.

Reading from or writing to a panStream results in a run-time bounds check. Because of this bounds check, adding panStream primitives to a bytecode language could increase performance by allowing the elimination of redundant bounds checks.

G. Soft state

All soft state stored in a node is associated with individual code objects. Code objects can store small amounts of state (such as protocol variables) in static variables or static class fields, which persist for the code object's lifetime in the node. The Intel ix86 mobile object code system requires explicit calls to make software segments persistent by placing them in a code object's software segment ring; when container software segments (such as hash tables) are made persistent, the software segment resources they contain also become persistent. The Java mobile code system simply uses Java's garbage collection system to prevent software segments stored in fields or objects from being freed.

H. Node implementation

There are currently two PAN node implementations, which share most of their code. One runs as a user-space process on a UNIX-like operating system; communication with applications uses UNIX domain stream sockets, and messages are sent to other nodes using UDP/IP packets. The other is implemented as a loadable kernel module for the Linux 2.0 operating system; communication with applications uses a special socket type, and messages are sent to other nodes using a protocol layered on top of IP. The Linux kernel implementation uses only existing kernel interfaces and requires almost no changes to a stock Linux 2.0.32 kernel. It should be relatively straightforward to add additional implementations (for example within a BSD-derivative such as FreeBSD or even within WindowsNT or on top of an Exokernel [27]).

Code within the PAN node implementation can be platform independent or platform dependent. The platform-independent code manages and accesses memory and node resources through the unified software segment mechanism, loads and processes code objects, maintains a cache of code objects, creates capsule environments, and evaluates capsules within their environments. The platform-dependent code handles node addresses, communicates with applications and across the network, and dispatches capsules as they are received.

I. Network links

Nodes within a PAN network communicate with each other by sending capsules across stateless (and possibly unreliable) network links (called *netlinks*). All nodes in a PAN network are identified by unique 128-bit *node addresses*. Different underlying implementations are used for the network links, depending on whether the network is made up of user-space nodes (which communicate using UDP packets) or kernel nodes (which communicate using a special PAN protocol layered on top of IP). Since PAN doesn't yet support IPv6, only 48 bits (a 32 bit IP address plus a 16 bit UDP port number) of the node address are actually used.

When a node receives a capsule, across a network or application link, it calls the `accept` method of the capsule's code object, passing the capsule encapsulated as a panStream. When a code object sends a capsule to another node, it uses an interface that sends data starting at the current position of a software segment. These mechanisms avoid extraneous data copies.

The UNIX user-space implementation uses UDP sockets to send capsules between nodes using the `sendmsg` and `recvmsg` system calls. By using `iovecs`, chains of software segments can be sent and received, without the need to copy data to a separate buffer before passing it into kernel-space.

The Linux 2.0 kernel implementation sends capsules between nodes using a special PAN protocol layered on top of IP. This avoids both interference with other IP-based protocols and the overhead of having an existing protocol, such as UDP, encapsulate capsules. PAN wraps the sk_buffs that contain arriving packets with software segments, thereby giving code objects access to arriving data without copying it. If a code object wishes to pass a software segment back to the node for transmission (as happens in the common case when capsules just send themselves on towards their destinations), the node uses the sk_buff already contained within the software segment for retransmission. The software segment is copied into a new sk_buff only if it is newly allocated by the capsule, or if its sk_buff is in the process of being transmitted by the kernel's networking layer. This greatly minimizes the number of times that capsule data needs to be copied.

In the common case when a capsule fits within an unfragmented IP packet and only forwards itself, the data in the capsule is never copied between its receipt by the network interface and its retransmission through another network interface. For some network drivers, untouched capsule data may never even be brought into the CPU or any caches.

Nodes send capsules to the network by prepending an appropriate IP header before calling the kernel `ip_forward` function. PAN nodes let the IP layer deal with fragmenting and unfragmenting packets that are larger than the maximum size the underlying network interface can handle. However, the maximum capsule size is constrained by the maximum size of an IP packet. At some point it may make sense to allow code objects to query nodes as to the largest size capsule that can be sent without fragmentation. This would allow code objects to control capsule sizes in order to improve performance—PAN capsules need to be reassembled and fragmented at each node they travel through, while IP routers don't typically incur this cost unless they need to apply firewall rules.

IV. Experimental Results

Experiments were conducted to compare the performance of the two implementations of a PAN node (in user-space and kernel-space) with each other and with a Linux workstation acting as a router. Both the latency incurred by the active network system and the total throughput of the system were measured. The experiments demonstrate that an in-kernel active network node executing native object code can forward capsules fast enough to saturate a 100Mbps Fast Ethernet network with 1500 byte packets while taking only 13 percent longer than a traditional node to process each packet.

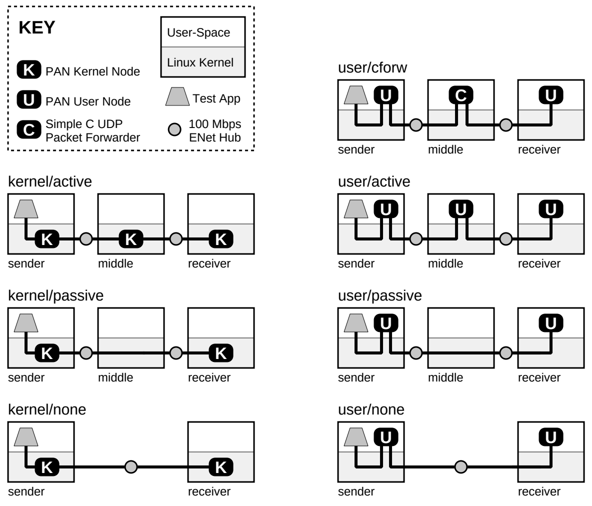

Fig. 2. Seven different testbed network configurations are used for measuring the performance of the user-space and kernel-space PAN implementation. During the latency experiments, the user-space configuration is used with both the native ix86 object code and Java mobile code systems.

The experiments using native Intel ix86 object code provide a good upperbound on performance, since the code is native to the processor and provides little safety; as a result, it incurs little interpretation, compilation, safety, and security overhead. Additional experiments performed by using a JavaVM to evaluate capsules indicate the sorts of performance costs that can be incurred by a mobile code system.

A. Experimental setup

All experiments were performed on a testbed network consisting of three Linux workstations connected by dedicated 100 Mbps Fast Ethernet network links. In all experiments, a client application runs on a *sender* node and sends capsules towards a *receiver* node, which evaluates the capsules and optionally sends back a response. The sender and receiver may be connected directly or through a third *middle* node.

Each machine contains a 200MHz Intel PentiumPro processor, an ISA/PCI motherboard with a 440FX chipset, and 64MB of RAM. The sender and receiver nodes each contain a single DEC DS21140 Tulip-based SMC EtherPower 10/100 network card running in 100 Mbs half-duplex mode. The middle node contains two Tulip network cards. The machines are connected by two Intel Express 100BaseTX hubs. Each machine runs the RedHat 4.2 distribution of Linux using an unmodified kernel from the `kernel-2.0.32-1` package and Version 0.79 of Donald Becker's Tulip ethernet driver.

Seven different testbed configurations are used for the tests (see Figure 2). In the three *kernel/∗* configurations, the sender and receiver nodes run the PAN kernel implementation using native ix86 code objects; in the four *user/∗* configurations, they run the user-space PAN implementation, also using native ix86 code objects. In the *∗/active* configurations, the middle node runs the same PAN implementation as the sender and receiver, and it processes all capsules sent between the sender and receiver. In the *∗/passive* configurations, the middle node uses the kernel's IP forwarding to forward packets between the sender and receiver. The *∗/none* configurations do not contain a middle node and directly connect the sender and receiver. Finally, in the *user/cforw* configuration, the middle node runs a simple user-space UDP forwarder written in C. Comparing the performance of the different configurations provides insight into where various overheads come from.

The UDP forwarder uses the same system calls as the user-space PAN implementation, but doesn't do any active processing. This configuration gives insight into how much of the cost of user-space PAN is due to active processing as opposed to how much is due to the overhead of transferring data to and from user-space.

The PAN kernel implementation uses most of the same code path as Linux IP forwarding, but adds hooks for capsules to perform active processing, which allow them to take actions other than just forwarding themselves towards a destination. As a result, the base case performance of PAN will always have slightly lower performance than standard IP forwarding.

All *user/∗* configurations were also tested using a JavaVM to execute code objects. The JavaVM used was a 1998.12.29 snapshot of Transvirtual's Kaffe OpenVM that was configured to perform JIT translation of JavaVM bytecode.

In addition to the seven basic configurations, latency experiments were also run in a *kernel/activecopy* configuration that is identical to the *kernel/active* configuration except that it uses a slightly modified PAN kernel node for the middle node alone. Normally, PAN doesn't need to copy or touch the contents of capsules, except for capsule headers. The middle node in the *kernel/activecopy* configuration copies all data before sending it in order to measure how much is actually gained by PAN's memory management system.

To see how packet size affects performance, all experiments were run using packets that contain 128, 156, 512, 1024, 1500, 1504, 2048, and 8192 bytes. In experiments with the PAN kernel implementation, these sizes include the 20 byte IP header. In experiments with the user-space implementation, they include both the 20 byte IP header and the 8 byte UDP header. This avoids overly penalizing the user-space implementation for having a larger header size. Because 1500 bytes is the MTU of Ethernet, the kernel's IP layer fragments and later reassembles any packets larger than 1500 bytes. Measurements were taken with both 1500 and 1504 byte packets in order to better see the discontinuity caused by packet fragmentation. Note that the Linux kernel needs to copy packet data in order to fragment packets.

The current implementation of PAN is essentially untuned. It should be possible to achieve substantial performance gains by analyzing and reducing existing performance bottlenecks.

B. Measuring latency

Measuring and comparing the latencies between PAN nodes in different configurations provides insight into the sources of performance gains. By comparing the times of the *∗/active* and *∗/passive* configurations, it is possible to determine how much more time is spent performing active processing than simple IP forwarding. The various *∗/active* configurations show how particular design decisions affect performance: the *kernel/active* configuration provides a baseline for what can be achieved, the *kernel/activecopy* configuration demonstrates the cost of performing copies within the kernel, the *user/cforw* configuration helps demonstrate the cost of bringing data into user-space, the *user/active* configuration demonstrates the cost of running a node in user-space, and the *java-∗/active* configurations demonstrate the additional cost of using a Java mobile code system rather than native ix86 object code.

Latency was measured for an active ping application, which behaves similarly to the UNIX *ping* utility that uses ICMP ECHO responses to measure round-trip times. Active ping works by sending a ping capsule from a sender to a receiver. On reaching the receiver, the ping capsule sets a state variable indicating that it has reached its destination and then sends itself back towards the sender. When it reaches the source, the capsule uses the *portmapper* code object to deliver the ping response back to the application. In order to obtain more accurate measurements with lower variances, the ping capsule timestamps itself within the sender node and then calculates the round trip time on arriving back at the sender node. Each ping capsule consists of a 100 byte header followed by a data body that fills up the rest of the packet. The *java-∗/∗* experiments run in user-space and use *ping* and *portmapper* code objects that were translated by hand from their C counterparts.

In addition to being a benchmark, the active ping application shows how an active network can provide functionality that is special-cased in traditional networks (ICMP ECHO responses) without requiring any special support in the network infrastructure.

For each testbed configuration and packet size, 10,000 ping capsules were sent and their round trip times averaged. These experiments were repeated three times each, and the median of the three trials was taken and used. Before any measurements were taken, the code objects needed by ping were loaded into all of the network nodes.

Figure 3 shows the end-to-end round trip times of ping capsules under different network configurations. The discontinuity between 1500 and 1504 bytes is due to packets being fragmented.

In Figure 4, the per-capsule latency incurred by the middle forwarding node is shown. This latency is calculated by halving the difference between the round trip times for the *none* configurations and the corresponding *active* and *passive* configurations. With 128 byte packets, just passive IP forwarding takes about 50 microseconds. For 1500 byte packets the passive forwarding time increases to about 160 microseconds.

Figure 5 shows the processing overhead incurred by various configurations relative to either the *passive* or *cforw* configuration. Because the PAN kernel implementation doesn't touch the contents of capsules, the overhead of *kernel/active* relative to *kernel/passive* remains around a constant 20 microseconds, regardless of packet size.

This can be contrasted to the *kernel/activecopy* configuration which has an overhead that grows with packet size. For 1500 byte packets, the overhead of *activecopy* relative to passive forwarding grows to over 45 microseconds. Thus, the cost of just copying a packet (about 25 microseconds for a 1500 byte packet) is larger than the entire overhead of active capsule processing. The cost of copying data actually drops for packets larger than the MTU because the overheads are computed relative to the passive forwarding node which is incurring the cost of dealing with IP fragmentation.

The user-space PAN implementation has a higher overhead than the kernel implementation, especially for large packet sizes. Because packets are copied to and from user-space, the shape of the overhead curves for the user-space implementation and the UDP forwarder are similar to the shape of the curve for the *kernel/activecopy* configuration. For 128 byte packets, the user-space implementation has an overhead relative to the IP forwarder of about 83 microseconds, or over four times the overhead of the kernel implementation. For 1500 byte packets, this overhead increases to almost 130 microseconds, or six times the overhead of the kernel implementation. Even when compared against the user-space UDP forwarder, the user-space implementation still has a considerably higher overhead than the PAN kernel implementation. It is not immediately clear why this is the case.

The performance overhead imposed by using a JavaVM to process capsules is substantial and dwarfs the overhead caused by copying capsule data across the kernel boundary. For 128 byte packets, the user-space implementation has an overhead relative to the IP forwarder of about 398 microseconds, which is 4.8 times that of the native code user-space version and over 21 times the overhead of the kernel-space version. Much of this cost seems to be coming from Java's memory management — the garbage collector halts capsule processing for hundreds of microseconds each time it runs. The cost of garbage collection is amortized into the *java-GC-user/∗* numbers shown in the figures. The *java-noGC-user/∗* numbers do not include the cost of the garbage collector running (capsules with round-trip-times of over 150ms are ignored when generating averages since it is likely that the garbage collector ran while the capsules were being processed). Even with the cost of garbage collection factored out, the processing time for 128 byte packets is still 266 microseconds (or over three times that of the native code user-space version).

To better understand the cost of using the Java mobile code system, timing instructions were hand-inserted to profile where time was being spent during capsule execution on the middle node. This makes it possible to directly compare the cost of executing capsules in two mobile code systems while ignoring the costs of crossing into user-space. The inserted timing instructions return the state of the processor's cycle counter. This set of experiments was performed using 1500 byte packets. Whereas only about 13 microseconds are spent in user-space setting up the environment for and executing the native ix86 capsule code, 187 microseconds are spent doing the same for the Java capsule code. Of this, 45 microseconds are spent creating the capsule's execution environment (which primarily involves creating a number of wrapper objects), 72 microseconds are spent reading in the capsule's header (which involves creating a number of objects), 29 microseconds are spent writing out modifications to the capsule header, and 11 microseconds are spent in user-space during the method call to forward the capsule towards its destination. Other parts of the capsule code run in a total of 30 microseconds. For reference, the instantiation of a single Object in Java takes about 5.5 microseconds. That it takes more

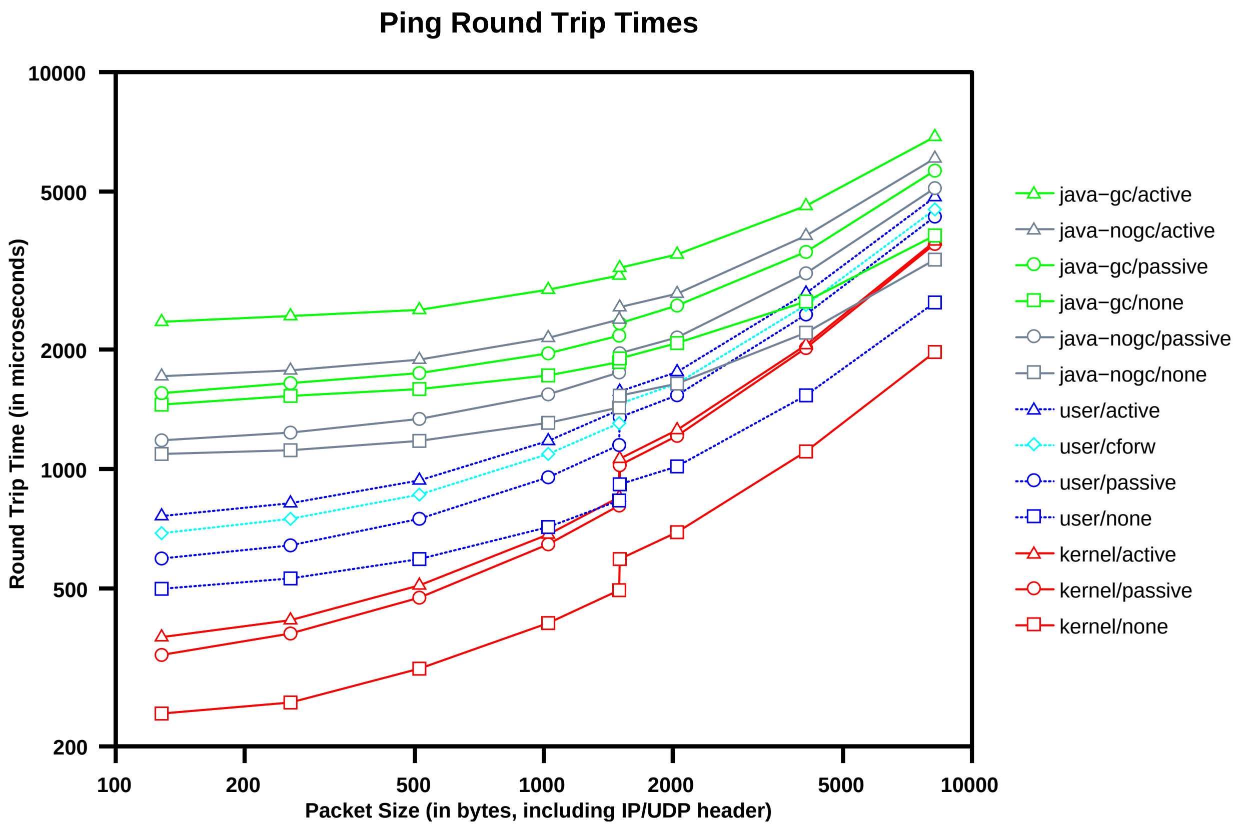

Fig. 3. End-to-end ping round trip times. Both axes use a logarithmic scale.

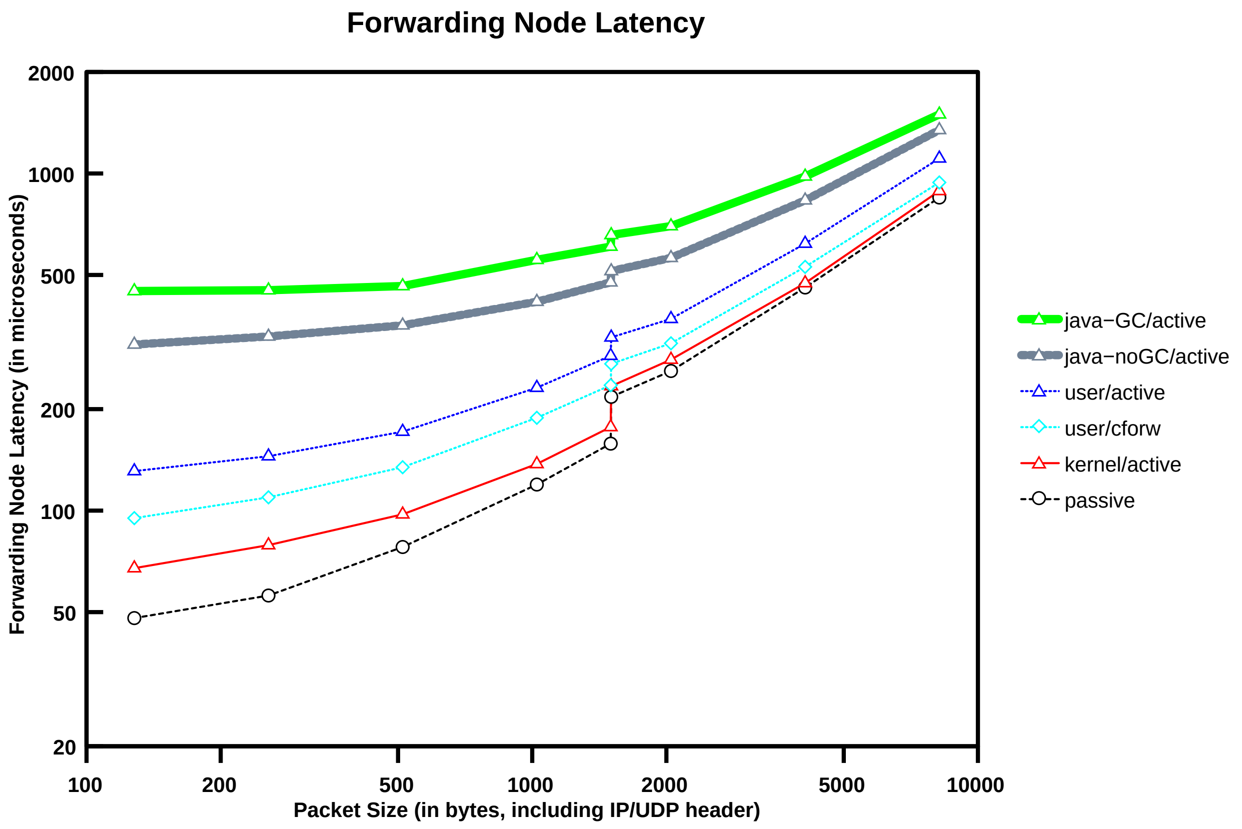

Fig. 4. Latency per capsule incurred by forwarding node. The horizontal axis uses a logarithmic scale. All ∗*/passive* configurations incur similar latencies.

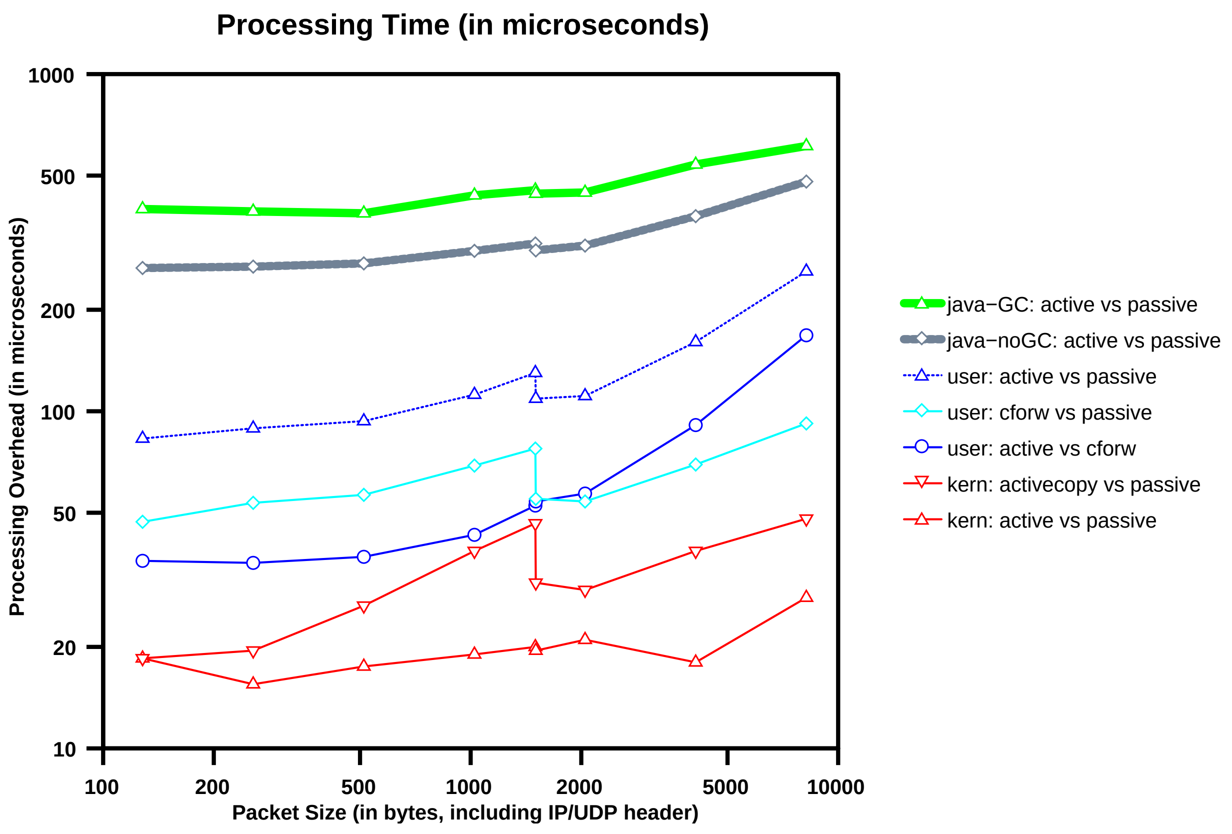

Fig. 5. Overhead (in microseconds) for forwarding each capsule, relative to passive or C forwarder. Both axes use a logarithmic scale.

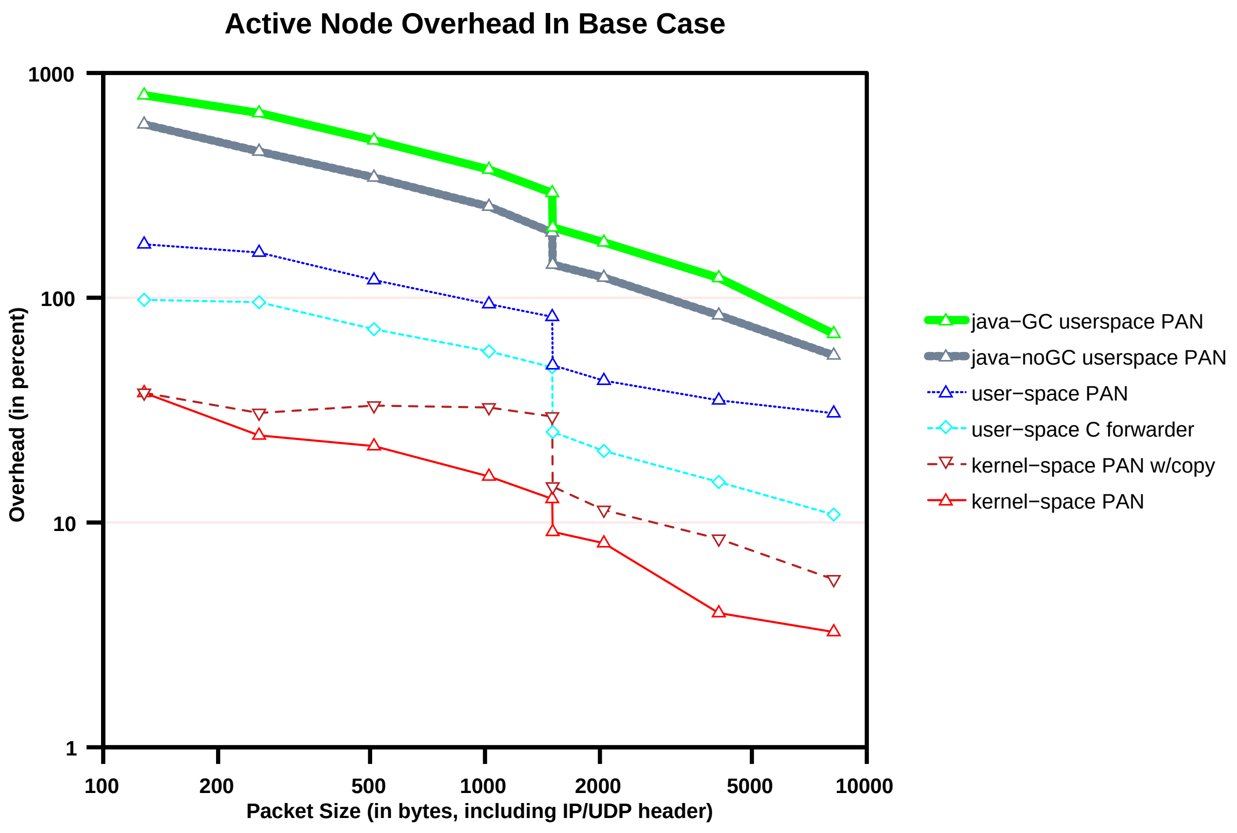

Fig. 6. Percent overhead for forwarding, relative to passive forwarder. Both axes use a logarithmic scale.

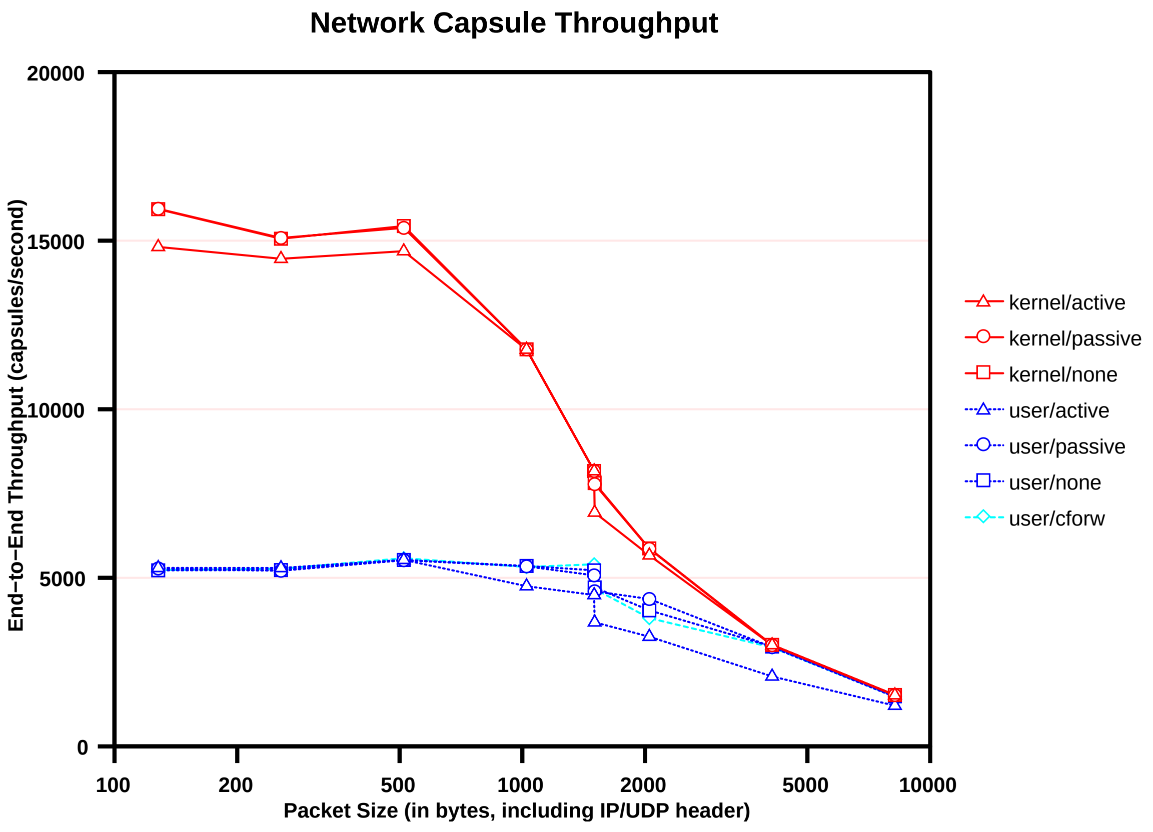

Fig. 7. Flood throughput in capsules per second. The horizontal axis uses a logarithmic scale.

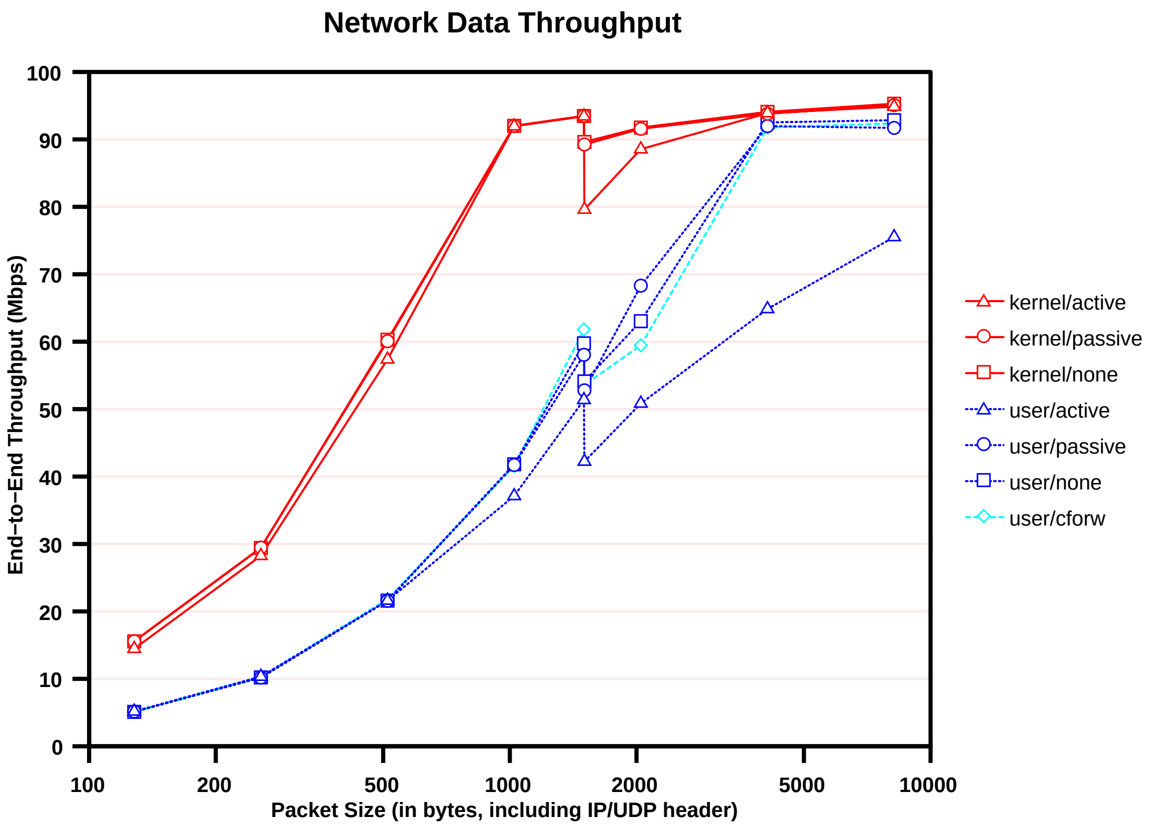

Fig. 8. Flood throughput in megabits per second. The horizontal axis uses a logarithmic scale.

time for three object creations than for the execution of the entire native ix86 capsule code (13 microseconds) indicates that substantial improvements are needed in the performance of Java mobile code systems.

Further evaluation is needed to fully understand the impact that various aspects of the Java mobile code system have on the performance of PAN. Additionally, it is possible that other Java virtual machines may offer better performance than the Kaffe virtual machine used for these experiments. A number of Java virtual machines are compared using microbenchmarks in [13]. Unfortunately, these microbenchmarks don't compare the costs of object creation or garbage collection, both of which have a significant effect on the performance of capsules running within PAN.

Finally, Figure 6 shows the active overheads as percentages relative to the passive IP forwarder. It shows that the relative cost of using an active network drops substantially as packet size grows. This happens because the passive IP forwarding latency increases with packet size while the active processing latency remains constant. At its worst, the kernel node has a 38% overhead for 128 byte packets. For 1500 byte packets, the kernel-space node incurs only a 13% overhead. For 8192 byte packets, this overhead drops to 3%.

C. Measuring throughput

The throughput of PAN nodes was measured using an active flood application that pushes capsules across the network as rapidly as possible. The flood application first inserts a *flood* capsule into the source node with an indicator marking it as the start of a flow. This capsule forwards itself to the receiver node, where it creates an entry for the flow in soft state, keyed by the address and port number of the originating application, and containing the starting time of the flow. The flood application continues to send capsules towards the receiver, where they increment the count of received capsules in the soft state entry for the flow. At intervals specified in the starting capsule, the receiver sends status capsules to the source with the elapsed time since the start of the flow and the number of capsules received since that time. Updates are sent only periodically to minimize their effect on measurements. Each flood capsule contains a 100 byte header followed by a data body that fills up the rest of the packet.

Because flow control and reliable communications protocols have not yet been implemented on top of PAN, measuring throughput required care to avoid packet loss. Experiments were performed by sending as many packets as could be sent without experiencing significant packet loss, which would have interfered with the measurements. As for latency, the median of three measurements was taken for each network configuration and packet size. Also, to eliminate code object load time as a factor, all code objects were loaded into all nodes before taking any measurements.

Figures 7 and 8 show the measured throughput in both capsules per second and megabits per second.

For smaller packets, the measurements of the user-space implementation are limited by the rate at which the sender was able to insert capsules into the network. Therefore it does not provide a good indication of the potential throughput of the middle forwarding node. This is indicated by the measurements showing that the *user/passive* and *kernel/passive* configurations had significantly different throughputs for packets smaller than about 4096 bytes. In a system in which end nodes and network collisions are not limiting factors, both should have had the same throughputs.

For both the kernel and user-space nodes, throughput for small capsules is limited by the time required for the node to process the capsules. For the kernel node, these processing times are on the same order as those measured in the latency experiments. For capsules larger than about 800 bytes, the throughput of over 90 Mbps is high enough to effectively saturate a Fast Ethernet network. Prior to this saturation, a small performance difference of under 8% can be seen between the throughput of the PAN node and IP forwarding. After the network saturates, both configurations have very similar throughputs.

V. Conclusions

The current PAN implementation achieves high performance and interoperability, but performs only minimal resource management and does not yet fully address the safety and security issues. Experiments performed with PAN indicate that a capsule-based active network can provide significant flexibility with only a small performance overhead compared to a traditional network node. The untuned kernel implementation of PAN is able to saturate a 100 Mbps Fast Ethernet with 1500 byte capsules with an overhead of 13% compared to the time a traditional network node takes to process capsules.

The most significant challenge to simultaneously providing high performance, safety, security, and interoperability in an active network is the lack of lightweight and embeddable high-performance mobile code systems. It may be worthwhile to develop a mobile code system just for this purpose — many of the features of the Java virtual machine, such as inheritance, are not required for implementing active network code objects.

Given a mobile code system capable of generating high-quality native code, achieving high performance is a matter of careful design. The most important aspects of the implementation involve running within the kernel, not copying or touching capsule data whenever possible, and evaluating capsules using code objects that have been converted into native executable code at load time.

Active networks also have the potential to improve overall network performance by minimizing redundant or superfluous network traffic. The flexible nature of PAN should help improve end-to-end performance by giving applications as much control as possible over what happens to their data within the network.

Acknowledgments

We would like to thank the members of the MIT Student Information Processing Board and the members of the PDOS and SDS groups of the MIT Laboratory for Computer Science for providing invaluable suggestions and comments since the start of this project. We are particularly indebted to David Wetherall for his guidance and for his work on ANTS, which was the primary inspiration for this work. We would also like to thank the anonymous OpenArch referees, whose feedback allowed us to improve the quality of this paper.

REFERENCES

[1] David L. Tennenhouse and David J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, vol. 26, no. 2, pp. 5–18, April 1996.

[2] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden, "A Survey of Active Network Research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, January 1997.

[3] Giovanna DiMarzo, Murhimanya Muhugusa, Christian Tschudin, and Jürgen Harms, "The Messenger Paradigm and its Implications on Distributed Systems," in *Proceedings of the ICC'95 Workshop on Intelligent Computer Communication*, June 1995, pp. 79–94.

[4] David J. Wetherall and David L. Tennenhouse, "The ACTIVE IP Option," in *Proceedings of the 7th ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[5] David J. Wetherall, John V. Guttag, and David L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *IEEE OPENARCH'98*, San Francisco, California, April 1998.

[6] Li-Wei H. Lehman, Stephen J. Garland, and David L. Tennenhouse, "Active Reliable Multicast," in *IEEE Infocom 1998*, San Francisco, California, April 1998.

[7] Ulana Legedza, David J. Wetherall, and John Guttag, "Improving The Performance of Distributed Applications Using Active Networks," in *IEEE Infocom 1998*, San Francisco, California, April 1998.

[8] Alden W. Jackson and Craig Partridge, "Smart Packets, A DARPA-Funded Research Project," Slides from 2nd Active Nets Workshop, March 1997.

[9] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura, "Implementation of an Active Networking Architecture," Networking and Telecommunications Group, College of Computing, Georgia Tech, White paper presented at Gigabit Switch Technology Workshop, Washington University, St. Louis, July 1996.

[10] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith, "The SwitchWare Active Network Architecture," *IEEE Network Magazine*, vol. 12, no. 3, May/June 1998.

[11] 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*. 1998, pp. 86–93, ACM.

[12] Pawan Goyal and Gísli Hjálmtýsson, "PRONTO: A Platform for Programmable Networking," Slides from talk at IEEE OPENSIG'98, October 1998.

[13] John Hartman, Larry Peterson, Andy Bavier, Peter Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Todd Proebsting, and Oliver Spatscheck, "Joust: A Platform for Liquid Software," *To appear in IEEE Computer*, 1999.

[14] John Hartman, Udi Manber, Larry Peterson, and Todd Proebsting, "Liquid Software: A New Paradigm for Networked Systems," Tech. Rep. TR 96-11, The University of Arizona Department of Computer Science, November 1996.

[15] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting., "Scout: A Communications-Oriented Operating System," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, Washington, May 1995.

[16] George C. Necula and Peter Lee, "Safe Kernel Extensions Without Run-Time Checking," in *Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996, pp. 229–243.

[17] James Gosling, "Java: Past, Present, and Future," Slides from talk at Software Development 98, February 1998.

[18] Erik Nygren, "The Design and Implementation of a High-Performance Active Network Node," M.S. thesis, Massachusetts Institute of Technology, February 1998.

[19] Sun Microsystems, Mountain View, CA, *The Java Virtual Machine Specification*, September 1996.

[20] Sun Microsystems, Mountain View, CA, *Java Native Interface Specification*, Release 1.1 edition, May 1997.

[21] "FIPS 180-1. Secure Hash Standard," U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, Virgnia, April 1995.

[22] Ron Rivest, "The MD5 Message-Digest Algorithm," Network Working Group Request for Comments, April 1992, Internet RFC 1321.

[23] Jerome H. Saltzer, David P. Reed, and David D. Clark, "End-to-end Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 2, pp. 277–286, November 1984.

[24] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham, "Efficient Software-Based Fault Isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, North Carolina, December 1993, pp. 203–216.

[25] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[26] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, and Dirk Verworner, *Linux Kernel Internals*, Addison-Wesley, Harlow, England, second edition, 1998.

[27] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, December 1995, pp. 251–266.