Appeared in *2003 IEEE Conference on Open Architectures and Network Programming Proceedings (OPENARCH 2003)*, pages 97–106, San Francisco, CA, April 4–5, 2003.

Bees: A Secure, Resource-Controlled, Java-Based Execution Environment

Tim Stack Eric Eide Jay Lepreau
University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

{stack,eeide,lepreau}@cs.utah.edu http://www.cs.utah.edu/flux/

***Abstract*— Mobile code makes it possible for users to define the processing and protocols used to communicate with a remote node, while still allowing the remote administrator to set the terms of interaction with that node. However, mobile code cannot do anything useful without a rich execution environment, and no administrator would install a rich environment that did not also provide strict controls over the resources consumed and accessed by the mobile code.**

Based on our experience with ANTS, we have developed Bees, an execution environment that provides better security, fine-grained control over capsule propagation, simple composition of active protocols, and a more flexible mechanism for interacting with end-user programs. Bees' security comes from a flexible authentication and authorization mechanism, capability-based access to privileged resources, and integration with our custom virtual machine that provides isolation, termination, and resource control. The enhancements to the mobile code environment make it possible to compose a protocol with a number of "helper" protocols. In addition, mobile code can now interact naturally with end-user programs, making it possible to communicate with legacy applications. We believe that these features offer significant improvements over the ANTS execution environment and create a more viable platform for active applications.

I. Introduction

Mobile code opens a door for users to execute custom programs on remote machines, enabling such technologies as active networks and mobile agents. In the case of active networks, an *active protocol* provides a means for end nodes to communicate in a customized fashion. For example, an active multicast protocol can make application-specific decisions about whether or not to propagate a packet. In the case of agents, code is distributed to end points in order to digest and act upon data available at a node. For instance, a stock monitoring agent could migrate to a well-connected node that receives market information, wait for a stock to hit a certain mark, purchase a number of shares, and terminate. Safely executing these protocols and agents is the job of an *execution environment*: the software on a node that most directly "contains" mobile code. As the stationary basis of a mobile code system, an execution environment (EE) is tasked with providing an expressive programming interface to mobile code, while at the same time protecting itself and other node resources from accidental or intentional abuse.

One aspect of expressiveness that directly affects protection is the amount of pressure that mobile code can exert on a node's resources. Many active network systems, such as PLAN [1] and SNAP [2], restrict the resources consumed by a protocol to those required to forward a single packet. We refer to these systems as providing "lightweight" environments. This model is very similar to existing IP networks in which all the parties and their packets are treated equally. As a result, accounting and resource control covering all packets in a flow can be avoided. This simplicity limits the possible set of protocols, however, since resources can only be consumed while executing the forwarding loop. At the other end of the spectrum are the "heavyweight" environments that allow mobile code to establish long-running programs that perform tasks independent of packet processing. The heavyweight option affords a greater amount of flexibility, at the cost of the added complexity required to perform the necessary authentication, protection, accounting, and resource control. Mobile agent environments use this model, and it is also the target of our work.

Our research in active networking began with Janos [3], the Java-oriented Active Network Operating System. The goal of the Janos project was to produce a resource-controlled operating system that could safely execute untrusted Java byte code. To leverage existing work, we chose the Active Network Transport System (ANTS) [4] to provide the execution environment for mobile code.

ANTS was envisioned as a lightweight environment that would execute Java code produced by anyone. Unfortunately, as the Janos project progressed, it became clear to us that the desires of our users (both internal and external) were not a good match with the intended uses of ANTS. For example, many users wanted to give special privileges to their active protocols. However, even privileges as simple as opening a file could not be allowed because there was no way to authenticate the mobile code making a request. The missing security features of ANTS were especially detrimental in the context of Janos: they prevented us from taking advantage of the resource controls that Janos provides. The lack of authentication and

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreement F30602–99–1–0503. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

© 2003 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

authorization services meant that there was no way for users to acquire additional CPU or network resources, resulting in all network flows being treated equally. Performance was also an issue; for example, ANTS always added an extra layer of routing to any packet transmission. Maintaining ANTS "in the field" exposed problems with the division of responsibilities between end-user programs and the active protocols. Finally, some of the applications we wished to deploy required the extra flexibility inherent in heavyweight execution environments — support that was missing in ANTS. Ultimately, Janos with ANTS simply could not meet the requirements of the applications we wished to deploy.

As a result of our experience, we set a goal to produce a more flexible Java-based execution environment that provided the mechanisms needed to deploy a secure and usable system. Starting with the simple ANTS APIs and programming model, we addressed the problems we had encountered in authoring protocols and administering active nodes. Our new system fills in the missing security features by employing a capability-based security model. Since Java is a major component of Janos, its inherent type safety eased the implementation of such a system. On top of capabilities, a flexible authentication and authorization methodology was defined, making it possible for downloaded code to gain access to any number of capabilities. Cryptography-based protections are not built into the core of the system: instead, the cryptographic code and objects are made available to the mobile code to use as needed. Protocols are able to use a simple form of composition, so that a primary protocol can easily make use of the services provided by a "helper." Finally, end-user applications are able to interact naturally with an active protocol and not have to deal with untrusted Java code.

We called the resulting system *Bees*. Overall, Bees foregoes built-in functionality in order to create a dynamic and reconfigurable environment. Atop the basic core, Bees provides *services* that an administrator can install to facilitate node discovery, authentication, and authorization. The design of the Bees-enabled Janos system builds on concepts previously explored by others, including the ALIEN architecture [5] and SANE environment [6] for structuring secure active networking systems in layers, the RCANE environment [7] for resource control, and the KaffeOS system [8] for managing multiple process-like entities within the context of a single virtual machine. A primary contribution of Bees is to implement these concepts in the context of an ANTS-like execution environment and make them available to users in combination with strong security (Section III) and novel features for the propagation (Section IV), assembly (Section V), and endpoint connection (Section VI) of Java-based active protocols. We demonstrate Bees with an example application (Section VII) and present initial performance results (Section VIII). In sum, Bees offers significant improvements over the ANTS execution environment by providing new and necessary features to users and administrators while also retaining much of ANTS' programming model and simplicity.

II. Background and Design Overview

In this section we give some background followed by a high-level overview of the major components of the Bees runtime.

A. ANTS

Bees is closely related to the Active Network Transport System (ANTS) [4]. ANTS was developed as a "lightweight" environment for active protocol code written in Java and found great success in the active network community. The runtime introduced the notion of an active packet, or *capsule*. A capsule is made up of user data and a unique identifier that refers to a class that must be used to process the capsule. When a node receives a capsule with an unknown identifier, ANTS downloads the appropriate bundle of capsule classes, called a *protocol*, from the previous hop. Once a capsule and its code have reached a node, the protocol can place node-resident state in a cache and perform whatever basic computation is needed to forward the packet. Security is limited to a simple access control list for locally started protocols, restricted access to Java classes, and the general assurance that Java code is too slow to saturate a network link.

However, it could be argued that ANTS presents an environment to mobile code that might be classified as a "middleweight." The system shares many of the traits of lightweight architectures, but also allows for node-resident state as seen in a heavyweight architecture. Unfortunately, this configuration seems vulnerable to an effect similar to thrashing in virtual memory systems, which we refer to as "protocol thrashing." A node experiences thrashing whenever the number of protocols being used exceeds the number of protocols it can support at any one time. While this does not adversely affect the node itself, any legitimate protocols experience a severe degradation of performance. This problem could be avoided by correlating resource usage to a single packet. As a result, the resources in use at the node by the protocol are the same before and after processing. This problem could be addressed by reworking ANTS towards one end of the spectrum or the other. Creating a lightweight ANTS would involve eliminating any possibilities for node-resident state being created, such as objects referenced by static fields. With these changes, the node would only have to pay the constant price of caching protocol code. However, except for offering a familiar Java-based environment to the programmer, there would be no real benefit over the PLAN and SNAP systems [1], [2]. The alternative is an environment that follows the heavyweight model, which is the approach we have taken with Bees.

B. Janos

Controlling the resources consumed by mobile code is a challenge faced by all execution environments. In a lightweight environment, resources are only consumed while forwarding a packet. The forwarding loop is dynamically limited or statically analyzed to control CPU usage. Memory is limited to the buffer holding the packet and some scratch space. In contrast, a heavyweight environment allows mobile code to consume resources at an unknown rate for an unknown length

of time. As a result, the environment must isolate the effects of an active protocol's resource usage and provide a method for terminating uncooperative protocols.

In the case of Bees, resource controls and asynchronous termination are built upon the facilities provided by Janos [3], the Java-oriented Active Network Operating System. Janos, in particular the JanosVM [9], provides a means to create isolated Java processes that coexist within a single virtual machine. The JanosVM accomplishes this by leveraging the type safety inherent in the Java language to prevent one process from gaining unauthorized access to objects in another. Thus, the virtual machine can support multiple processes without the need for hardware enforcement. Furthermore, each process maintains separate garbage collection and finalizer threads to properly account for the use of these language services. Finally, a combination of the OSKit [10], Moab [11] (our NodeOS implementation), and the JanosVM provides controls over the amount memory, CPU, and network bandwidth consumed by each Java process.

In addition to its roles in process isolation and resource control, the JanosVM provides the ability to share Java classes and asynchronously terminate Java processes. The class sharing ability is derived from Tullmann's Alta [12] Java operating system and allows for well-formed groups of classes to be exported from one process and imported into several others. Thus, Bees is able to reduce its total memory footprint by sharing its core classes with the processes housing the mobile code. In the JanosVM, process termination follows the rules laid down by Back's "red line" paper [13] and its use in KaffeOS [8], from which the JanosVM is derived. The "red line" is needed because the combination of class sharing and type safety makes it possible for user threads to execute system code that subsequently accesses shared system objects. Because of this possibility, system code must communicate to the virtual machine that asynchronous termination of a thread must be delayed until it has finished manipulating the shared object(s). Without this communication, the termination of an active protocol executing system code could result in critical objects being left in a damaged state, thus undermining the integrity of the entire system. Therefore, a "red line" is drawn between user processes and the kernel, and the crossing of this line serves as the signal to the JanosVM to delay termination of a protocol thread.

Bees receives an active protocol by starting a new Java process. After this point, any operations performed by or on behalf of the protocol are executed within the process and be properly accounted. In addition, to enforce resource management or security policies, Bees can reliably and asynchronously terminate protocols, and reclaim their resources.

C. Bees Overview

A Bees-based active node consists of end-user applications at the top, services that provide basic assistance when interacting with neighboring nodes, the Bees runtime, and finally the JanosVM. The application is any end-user program that wishes to use an active protocol to perform some task. For example, `wget` could use "active HTTP" to retrieve a file. Bees services provide facilities that are not a fixed part of the Bees runtime: example services are node discovery and the mobile code authentication and authorization agent, or "Auth Agent." Underlying the services, the Bees runtime provides the common ground on which the mobile code and the above systems interact. Finally, the JanosVM is used to isolate the instances of active protocols and control their resource consumption.

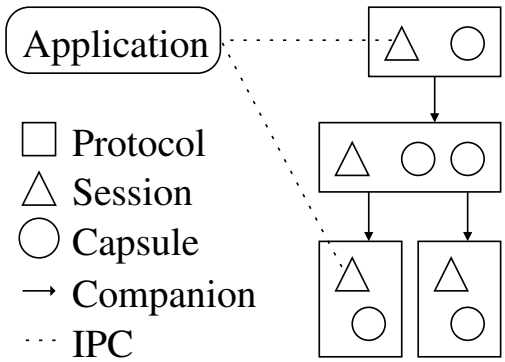

Fig. 1
A BEES PROTOCOL

The active protocols that a Bees node can execute are made up of the Java class files and any extra data needed by the implementation. The structure of the active protocols is shown in Figure 1 and is very similar to the ANTS design. A *capsule* is a network packet that is bound to a capsule class. A *protocol* is used to define the set of capsule classes, any support classes, and any extra data used by the protocol. Unlike ANTS, however, a protocol can contain two other elements. First, the protocol may add multiple helper protocols, called *companions*, which provide services for the main protocol. An example companion would be the system-provided protocol used to download active protocols from neighboring nodes. Next, the protocol can add a *session* class that acts as an adapter for translating end-user application requests into protocol activity. These extra elements and their benefits are described in detail in later sections.

As an example, we describe the components of a ping protocol that functions over a single hop. The `LivenessProtocol` is a service provided by Bees that is used to probe neighboring nodes to ensure they are still responsive. The protocol defines one capsule type, `PokeCapsule`, which is used to ping a neighbor. The packet handling code for this capsule simply checks a boolean value in the payload and takes one of two actions. If the boolean indicates a probe request, it is flipped and the capsule is sent back to the source. Otherwise, the capsule is a reply to a probe, indicating that the neighbor is still functioning and its timeout should be refreshed. In case a neighbor times out, the protocol will use its `LivenessSession` object to send a message to any registered applications informing them of the death of the neighbor.

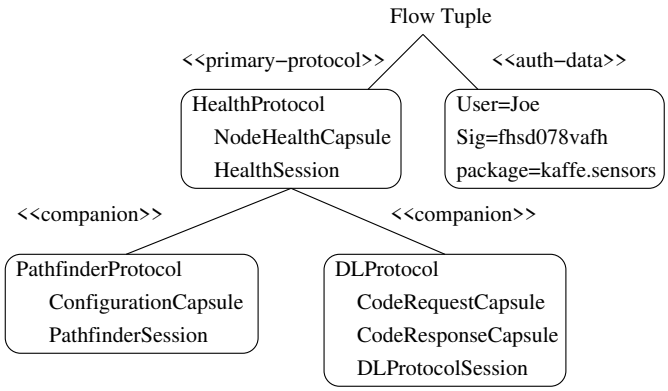

Fig. 2
A FLOW TUPLE

The `LivenessProtocol` also adds a configuration file and an extra class used to process this file. However, it does not contain any companions since it is expected to be pre-installed on the nodes. Hence, it does not make use of the built-in download protocol.

Active protocols by themselves do not constitute a mobile unit of code. The hierarchy of protocols must be combined with authentication data — typically, a digital signature — and serialized to form a *flow tuple*. The flow tuple can then be used by Bees to uniquely describe a flow of capsules in the network. For example, when a node receives an unknown capsule it will query the previous hop to map the unknown capsule identifier to the appropriate flow tuple. The Auth Agent can then use the authentication data to perform various checks before downloading the protocol. Because the tuple is basically a free-form collection of byte arrays, it can be used in a number of ways to communicate with the Auth Agent. For instance, a tuple could describe a set of privileges that should be given to the protocol when it is started on the node.

As an example, Figure 2 shows the structure of a tuple that describes the "HealthProtocol" as deployed by "Joe." The left side of the tree shows the hierarchy of protocols and their internal structure. This hierarchy is encoded in the tuple as the MD5 hashes of the class files or data chunks; the actual classes and data are not included. On the right side is Joe's authentication and authorization data. In this example, Joe identifies himself, requests that the protocol have access to the classes in the `kaffe.sensors` package, and provides a signature covering this data and the protocols. Now, when the tuple is presented to an Auth Agent on a node, it will check to see if Joe is a known principal and use the public key to verify the signature. If the signature is valid, the agent will check if Joe has access to the `kaffe.sensors` package, and if so, subsequently pass the capability to the protocol.

III. SECURITY

The security of a Bees node depends on a properly written Authentication and Authorization Agent, or *Auth Agent*. The Auth Agent is a node's first line of defense: it decides what protocols will and will not be started on the node. In addition, it will also make the decisions about what privileges the protocol will have while it is running on the node. Because much of this work is policy-specific and could involve any number of custom protocols, the agent is not a fixed part of the Bees infrastructure. However, we do provide a base for writing such applications, as well as two simple authenticators. The more interesting of the two is the "source based" agent that only allows flow tuples that match a known user and public key. In addition, the agent can grant any subset of the user's privileges to flows that have a valid signature.

Starting a new active protocol and its companions on a node is an operation carried out by Bees using the JanosVM. The hierarchy of protocols will be mapped to a JanosVM *team*: a process-like entity within the virtual machine, with its own heap, name space, garbage collector threads, and other resources. This mapping isolates the mobile code's name space to its own classes and those shared Bees and Java system classes. The mapping also serves to control and account for the resource usage of the protocols and defines the unit at which termination can occur.

Privileges in Bees are based on *capabilities* [14]. A capability is a Java object that provides access to a resource. Therefore, all who possess a reference to the object have access to the resource. For example, if the Auth Agent passes a capability to a file to an active protocol, it could read the contents of the file. Because Java offers type safety and fine grained control over what methods and fields in an object are visible, it is trivial to implement such a mechanism. Unfortunately, the standard Java runtime does not follow the capability-based security model. Thus, any classes that can create privileged objects, such as `java.io.FileInputStream`, need to be hidden and Bees alternative versions introduced.

Transferring capabilities between entities on a node is done using *envelopes*. These envelopes provide a means for wrapping, and optionally sealing, a capability for transfer to another JanosVM process running on the node. Sealing an envelope makes it impossible to open unless the code also possesses the appropriate unsealer. For example, instead of passing all the user's capabilities to an Auth Agent, the administrator could pass sealed envelopes that could only be opened by a signature separately sent to the newly created team in a capsule. As a result, the Auth Agent has less authority and requires less trust from the users and administrator.

Cryptographic mechanisms play a small role in the core of the Bees environment. As stated earlier, this is done to avoid creating dependencies on support protocols that should be easily replaceable. The system makes use of MD5 hashes to verify that downloaded code matches the hashes presented in the flow tuple. In addition, mobile code can request hop-by-hop integrity checks be performed for a given neighbor node and symmetric key. Otherwise, the active protocol has the choice of whether or not to use the cryptographic primitives available in the Java Cryptography Extension (JCE) [15]. For example, a protocol could sign capsule data generated at a node using the node's private key capability. Of course, the

key capability is designed to not leak the key material and the signer object will automatically prepend a hash of the flow tuple to any signed data to bind the signature to that tuple.

IV. Capsule Propagation

Communication between nodes in a Bees-based active network is done by the active protocol propagating capsules from the source to the destination node or nodes. Unbounded propagation of capsules by malicious or buggy code could unnecessarily burden the network; on the other hand, placing arbitrary limits on what operations an active protocol can perform merely shifts the burden onto the protocol author. Therefore, we must first define what constitutes a valid sequence of events for propagating a capsule before deciding where and when to draw the line.

Because Bees follows the "heavyweight" model, mobile code expects to be able to perform whatever operations it wants with the resources it was given by the Auth Agent. However, using network bandwidth affects not only the local node, but also the destination node and the network infrastructure in between. Therefore, the use of this resource must be dependent on some external stimulus, such as the receipt of a capsule or a timer event. Once this "authorization event" has been received, the active protocol is then free to use a bounded amount of network bandwidth.

ANTS implemented the above semantics through a Time-To-Live (TTL) field in the header of every packet. The receipt of a capsule authorized a protocol to reuse the capsule by sending it to a neighbor or transforming it into another capsule type. When one of these operations is performed, the capsule's TTL field is decremented and checked for a value greater than zero. Unfortunately, this method does not convey the actual propagation characteristics of the capsule type, and requires the system to believe a value in the header that is hard to verify. In [16], Wetherall makes the argument that the TTL is an insufficient means of bounding resource consumption in an active network: in ANTS, a malicious protocol can use a capsule with a high TTL to generate and forward a very large number of "child" capsules. Finally, the simple TTL method complicates mobile code that does not follow the typical routing protocol model; such "atypical" applications include those based on token rings and agents that send capsules based on timer events.

The Bees system maintains the "decrement until zero" style of the TTL; however, it explicitly enumerates the set of operations in which a type of capsule can be used and the set of "authorization events" that refresh the limits. Unlike a per-packet TTL, the limits attached to a capsule are defined in the protocol description by the protocol author and are not taken from the packet. These limits are then bounded by a node's Auth Agent since they are part of the flow tuple. For example, a user with the appropriate privileges could install a protocol that generated thousands of capsules from a single one.

The current set of authorization events are:

- *Protocol Initialization*: the system generates a "boot" capsule that is evaluated after a protocol has been installed.
- *Timer*: a periodic timer event refreshes the capsule's limits. The minimum interval is specified in the protocol description. The description also specifies the global number of capsules that can be refreshed at any time.
- *Capsule Receipt*: a capsule was received from another node in the network. This event also sets the "source neighbor" value in the capsule so the system can differentiate between forwarding to a new node or back to the source.
- *Application Request*: the protocol session received a request for some action.

These events make the capsule available for use in one of the following operations:

- *Initial*: the capsule object is being forwarded to a neighbor node and was not originally received from a neighbor.
- *Forward*: the capsule object is being forwarded to a neighbor node that is different from the original source of the capsule.
- *Return*: the capsule object is being forwarded to the original source of the capsule.
- *Neighborhood*: the capsule object is being forwarded to a set of neighbors. This value is used when implementing a multicast protocol.
- *Group*: the capsule object is being added to a group of capsules that can then be sent to a neighbor.
- *Transform*: the capsule object is being transformed into another capsule type. Any source neighbor attached to this capsule will be passed on to the other capsule object. The limits for other transformations are also passed, thereby disallowing unbounded cycles.

As an example, a ping protocol would set the "return" limit of its `PingCapsule` type to one. When the protocol receives a ping capsule, the capsule object's limits will be refreshed. Upon forwarding of the capsule back to the source, the "return" limit will be decremented. Any additional attempts to send the capsule would be disallowed since all of its limits are at zero.

Access to the above operations are mediated through *neighbor* capabilities. A neighbor is used to send and receive capsules from a directly connected node. Furthermore, neighbors can be grouped together into a *neighborhood*, making it possible to limit operations like multicast, regardless of the number of destination neighbors.

Currently, all of the limits are reset when a capsule is refreshed from an event. However, this model could be made more explicit by resetting individual limits based on the type of event, thus providing an even clearer picture of the protocol's intentions.

Capsule propagation can also be limited by simply not allowing a protocol to send capsules to a neighbor. Because neighbors are capabilities in Bees, restricting the spread of capsules and their mobile code is a trivial matter. For example, the Auth Agent on an edge router can contain a protocol to the internal network by denying it neighbors on the outside. In addition, the agent could decide to revoke a neighbor from a

misbehaving or buggy protocol, perhaps through some secure communication with that neighbor's Auth Agent.

V. Companion Protocols

End-user programs use active protocols to accomplish their tasks and, in turn, active protocols need to use other protocols to accomplish their tasks. For example, an active protocol uses the system-provided `DLProtocol` to transfer itself between nodes. This form of composition is similar to groups of non-active network protocols, like IP, ARP, and BGP. In Bees, these helper or *companion* protocols are defined by the system or protocol authors. Protocols are then paired with companions to take advantage of their services.

Pairing a companion with a protocol results in the protocol gaining access to the companion's session object. The protocol can then use the session to make requests of the protocol or simply query it for information the companion generated independently. For example, Bees includes a user-level routing companion, called Pathfinder. This protocol continually updates routing information that can then be used by the main protocol when routing its capsules. This separation of tasks also makes it simple to select what companion protocols will and will not communicate with a neighbor. For instance, a protocol could tell its code-downloading companion not to communicate with a specific neighbor, thereby preventing the protocol itself from being downloaded to that neighboring node. In addition, this separation also makes it simple to use different companions implementing a common interface depending on which implementation is more appropriate for a given situation.

From the companion's point of view, pairing with a protocol results in the companion's capsules being coupled with the capsules or the byte arrays that make up the other protocol. This part of the pairing process also decides what extra hashes will be used to compute the companion capsule's identifier so it will be unique in the network. In the case of the system downloader, it will pair off with the byte arrays so that its request and response capsules match a particular flow and the bits to be transferred.

However, our experience with ANTS showed that mixing the identifiers is not always desirable and using two separate identifiers has some useful properties. When a capsule with an unknown identifier is received by an ANTS or Bees system, it must send back a request to map the capsule identifier to a flow tuple. This process is implemented by a system flow that sends a mapping request capsule with the identifier of the unknown capsule. The request is then dispatched to the user flow on the destination node. Finally, the reply is received by the requesting node and used to download more information about the flow. Because the identifier for the mapping request capsule and the unknown capsule were kept separate, the reply was handled naturally by the system flow. In addition, by dispatching the request directly to the user flow, the resources used in processing the request were properly accounted for. The most striking part of these events is that two distinct flows were able to exchange capsules, an operation normally denied by the Bees model. In this case, however, the cross-flow communication is allowed because the capsules are from the same protocol. The difference is that the protocol is a companion in one flow and is the top protocol in the other. Clearly, the ability to cross flow boundaries with a companion is a powerful and dangerous abstraction. Therefore, these flow-independent companions are currently restricted to system flows and companions.

Because a companion is also a protocol, it can have its own set of companions. For instance, the Pathfinder companion uses the system downloader since it is a user protocol and also needs to be transferred.

Besides pairing with a protocol, a companion can also be associated with a flow tuple. In this case the capsules are paired with the description's hash. Currently, this pairing is only used by Auth Agents to download an unknown flow tuple.

VI. Protocol Session

Instances of active protocols communicate with each other using capsules; however, there is also a need to communicate with end-user programs. Since Bees considers all mobile code to be hostile, there is a strict separation between end-user programs and the protocols they employ. Therefore, simply loading the mobile code into the application is not an option. In addition, applications not written in Java should not be excluded from interacting with mobile code. Our solution to this problem is to introduce a *protocol session* object that exchanges plain packets with any interested applications. For example, a Web browser that uses "active HTTP" would send a GET request to the session object, which would cause the active protocol to generate zero or more capsules to satisfy the request. Notice that the protocol does not present the application with an interface for sending and receiving specific capsules. Instead, the protocol presents an abstract interface suitable for use by end-user programs. This approach is similar to socket interfaces where a single `write` call can cause several TCP segments to be sent.

The protocol session also presents an opportunity for applications to download active protocols from a protocol server. This approach has the effect of insulating applications from changes to the internals of the mobile code since they must only understand the less volatile session interface. For example, a freshly booted node can download some active configuration protocol from the local router instead of requiring the latest protocol be installed on the node.

VII. Example Application

As a demonstration of the Bees system, we have implemented a hardware health monitoring agent. The goal of this agent is to spread across a network of PCs, continually read the motherboards' sensor information (e.g., temperature, fan speed, and power supply voltage), and report this data back to a central server. This application was chosen because it takes advantage of many of the facilities provided by Bees, has an existing counterpart in legacy style systems (healthd [17]),

and is more substantial than the standard ping and multicast protocols.

A health agent starts its life on a server node that wishes to receive the health reports. Determining information for routing capsules and spreading the agent to the surrounding nodes is performed by the Pathfinder companion protocol described previously (Section V). Pathfinder executes a spanning tree algorithm using the server as the root of the tree. As a consequence of Pathfinder's periodic broadcasts of configuration capsules to discover the tree, the other nodes on the network download and install the health agent. Once installed on a node, the agent begins sampling the sensors and sending report capsules toward the server. The use of a companion protocol here relieves the health agent from the burden of determining a route to the server for report packets generated at the edge nodes. In addition, separating out the Pathfinder protocol makes it trivial to reuse for agents that have a similar traffic model.

Because the agent must gain access to a privileged operation — reading the motherboard sensors — the agent must have a flow tuple with the appropriate credentials. In this case, the tuple would be used to request the following capabilities from each node's Auth Agent:

- the `kaffe.sensors` package and the classes within;
- the node's private key, used to sign health reports; and
- a clock to timestamp health reports.

These capabilities are passed to the agent when it is first started on a node. However, it is possible for an agent to spawn onto a node where it does not have these privileges, in which case it will simply act as a router for any other nodes. Fortunately, the inherent dynamic linking properties of Java make it trivial for the agent to continue despite the missing `kaffe.sensors` classes.

Agent termination in the network is designed to work from the inside out. Terminating the agent on the server node will cause the other health agents in the network to time out and self-terminate. This self-destruct mechanism works because Pathfinder can directly interact with the system-provided companion protocol used to deal with unknown capsules. In this case, Pathfinder tells the companion to ignore requests from the node representing the next hop to the server. Therefore, the agent will not continually respawn onto other nodes in the network, and the existing agents will time out since they have nowhere to send their reports.

As stated earlier, one of the reasons for developing a health monitoring agent was the ability to compare it against a non-active counterpart, the FreeBSD Healthd daemon [17]. Healthd is a good example of the issues that come up with a non-active system:

- A node's sensor data is only available to other machines on the network via a specific protocol.
- Changes to the protocol's behavior must be redistributed manually, requiring the intervention of a system administrator.
- The daemon and protocol are extremely simple, yet it has been exploited through a buffer overflow [17].
- The daemon generally runs with more privileges than it requires to perform its function.
- Access to the protocol can only be restricted to certain hosts in the network using TCP wrappers [18]. There is no authentication of requesting entities.

TABLE I
ZERO-BYTE PAYLOAD RELAYING USING UDP

Implementation	Packets/Second
C	48255
Java	24882
Java NodeOS	16164
Bees	11018
ANTS	6761

Another reason for choosing this application is that it takes advantage of Bees features that are not found in ANTS. The ability to leave mobile code running for weeks and months at a time, like a typical daemon, is a benefit of the "heavyweight" model. Bees' changes to capsule propagation limits makes it possible to generate packets based on timer events, instead of just packet receipt. Furthermore, since the propagation behavior is visible to the Auth Agent, it can set the upper and lower bounds for system parameters such as timer intervals. As already stated, the agent makes use of the companion protocols, such as Pathfinder, to separate concerns and gain control over the behavior of the system-provided downloader protocols. The addition of capabilities and the Auth Agent make it possible to gain access to a privileged resource, i.e., the sensors. Finally, the core health agent protocol is abstracted away by the protocol session, reducing the complexity of end-user programs that generate reports from the information gathered by the agent.

VIII. PERFORMANCE

We measured the performance of Bees using the Utah Emulab [19] facility. The test application was a simple packet relayer implemented for each layer of abstraction provided by the overall Janos system architecture. To enable direct comparison with the current release of ANTS [20], which relies on an underlying network stack for packet routing, we ran our tests atop Linux rather than atop a "bare hardware" configuration (that would include Moab [11], our C implementation of the NodeOS). The implementations of our test application included a C program using Linux system calls, a Java program running within the JanosVM and using the standard Java socket APIs, a program using our Java implementation of the NodeOS [21], a protocol written for Bees, and a protocol written for ANTS. The test setup was three nodes: a packet source, the relayer, and the destination. Each node consists of an 850 MHz Pentium III processor running RedHat Linux 7.1, JanosVM version 0.9.0, and Bees version 0.5.0 or ANTS version 2.0.4.

Table I presents the relay rate for an empty UDP packet or capsule in the five configurations. The difference between

the Java NodeOS and the regular Java client is the handoff needed to pass the packet to a user thread. This handoff is used to properly account for any packet processing time and, unfortunately, contributes to the slowdown in Java. Because ANTS and Bees do not set upper limits on the time taken to forward a packet, they must use the handoff to prevent protocols from denying service to other protocols. The difference between ANTS and Bees is partially due to the removal of the default routing lookup performed by ANTS. The rest is due to Bees allowing capsule classes to recycle their capsule objects, resulting in less garbage collection.

IX. Related Work

As described earlier, the design and development of Bees was motivated by our experiences in using ANTS, the Active Network Transport System by Wetherall et al. [4], as the execution environment for mobile code in our Janos active networking project [3]. Bees improves upon the widely used ANTS model in the particular areas of flexible protocol assembly and cooperation, security, and resource management. Other researchers have explored these concerns as well, both inside and outside the ANTS framework.

For instance, the focus of the CANEs system by Bhattacharjee [22] is on the modular assembly of network services such as active reliable multicast [23]. In CANEs, a service is composed by adding customized code (called "injected programs") at predetermined points in a underlying generic service. Depending on the node policy, the set of underlying programs may be fixed or variable. The underlying and injected programs communicate via shared variables. The model for protocol composition in Bees is similar, but perhaps more flexible: a Bees protocol attaches (arbitrary) companion protocols to itself dynamically, and communication takes place through protocol session Java objects. Bees protocol assembly is therefore less structured than that in CANEs — there is no fixed, two-level notion of an "underlying" program being extended at well-defined points by "injected" programs. On the other hand, Bees protocol communication can be more structured, through the use of OO abstractions.

In the area of security, Murphy et al. created SANTS [24], a version of ANTS that incorporates additional features for strong security. Like SANTS, our Bees system is careful to protect itself from mobile code; Bees further constrains mobile code through the use of capabilities and capability envelopes. Like SANTS, Bees offers cryptographic services to applications via the Java Cryptographic Extension [15]. Unlike SANTS, however, Bees does not provide authentication as a core service. Instead, authentication and authorization decisions in Bees are made by a separate agent. This software organization makes it easier for node administrators to tailor their systems to their needs, e.g., to favor ease of use in "friendly" settings, or to enforce strict policies in production systems. The default Bees Auth Agent does not use X.509v3 certificates or DNSSEC storage of credentials, but we may add these features in the future.

Security is also the primary focus of SANE, a non-ANTS-based execution environment developed by Alexander et al. [6] as part of the SwitchWare project. SANE addresses safety and security "from the ground up": based on a minimal set of assumptions, SANE can securely bootstrap an active node, establish trusted relationships with network peers, and load the higher-level parts of its execution environment for mobile code. Secure bootstrapping of Bees is outside the scope of our work, but the higher-level attributes of SANE are similar to parts of Bees. SANE uses a type-safe language and "module thinning" to control the functions that are made available to mobile code; Bees uses Java's type safety and name space controls to achieve similar results. (When finer-grain control is required, Bees manages specific resources such as handles to network neighbors using capabilities.) SANE and Bees are also similar in that both focus on ensuring the safety and security of individual nodes through controlled access to resources ("node safety"), but do not guarantee the safe and secure use of network resources as a whole ("network safety") [25].

Active networking systems that focus on language-based resource control include PLAN [1], SNAP [2], and PLAN-P [26]. By restricting the language for expressing mobile programs, these systems can enforce strong resource controls over mobile code; these guarantees, however, may come at a cost in expressivity (some useful programs cannot be written) and programmer productivity (the need to learn special languages). In contrast to these systems, Bees allows mobile programs to be written in Java. Memory, CPU, and network resource controls in Bees are provided with the help of the JanosVM [9], which provides multiple process-like entities within the context of a single virtual machine. (The JanosVM can in turn utilize the resource management features of the Janos NodeOS, called Moab [11].) The JanosVM is derived from KaffeOS [8], developed by Back et al., but tailored for use in an active network setting.

The lower layers of Janos are comparable to those of the RCANE system developed earlier by Menage [7]. They share many of the same design traits with the differences lying primarily in the location of components and their implementation. Instead of Java, RCANE uses the O'Caml language to provide portability and isolation of processes in a single address space. The RCANE loader, which is derived from the ALIEN [5] architecture, is similar to the Java class loader used by Bees to restrict the name spaces available to mobile code. The functionality of the "Core Switchlet" in ALIEN is split across the NodeOS and the Auth Agent in a Janos-based Bees system. The NodeOS provides the mechanisms for sending and receiving packets while the Auth Agent implements the policy using capabilities to the NodeOS services. Resource controls for RCANE are provided by the Nemesis operating system, while Janos uses a combination of the OSKit, the Moab NodeOS, and the JanosVM. Bees complements these lower layers by providing a higher level and coherent framework that is suitable for exposure to untrusted code. A "heavyweight" execution environment for handling mobile O'Caml code in RCANE might follow a design similar to that of Bees.

Outside the domain of active networking, there is continuing interest in using Java to deploy network-based services: existing platforms for this include Sun's Jini architecture [27], Gribble et al.'s MultiSpace system [28], and IBM's Aglets for mobile agents [29]. We believe that Bees can support the development of these kinds of middleware. For instance, the MultiSpace system requires that its underlying JVM "behave and perform like a miniature operating system" [28] to protect and isolate the various components of the system; Bees does this, and offers additional network infrastructure besides. Concerning Aglets, the Aglet programming model allows a Java mobile agent to package up its current state and transfer itself to another host in the system. Although this model of shipping code and dynamic state together differs from the Bees model, it would be straightforward to implement an Aglet-like system with the help of a companion protocol that serializes dynamic state and sends it to any nodes that an agent chooses to visit. Bees can also help to ensure the correct behavior of mobile agents. With Aglets, communication between agents on separate machines is accomplished by sending message objects, much as Bees flows communicate using capsules. However, these message objects lack the automatic dispatch based on type provided by Bees and, more importantly, their propagation is not bounded by the system. Bees' security and resource control features would be useful for mediating inter-agent communication, and for otherwise protecting systems against buggy or malicious agents.

X. Conclusion

Mobile code presents an opportunity to build flexible networks. Unfortunately, these properties are limited by the execution environments provided to run the mobile code. On the one hand, an environment must be able to perform a wide variety of functions, or it will not be used. On the other, the environment must be able to make strong guarantees to the node's administrator, else it will not be installed onto a node.

We have presented Bees, a safe and flexible execution environment for running mobile Java code. Originally based on the Active Network Transport System (ANTS), the Bees execution environment adds a number of enhancements that address many of the needs of mobile code authors and node administrators. These enhancements include better security, composable protocols, and flexibility when interacting with end-user applications. Security comes from capabilities that can be distributed to protocols through a flexible authentication and authorization mechanism. The system is aware of the JanosVM and can take advantage of the isolation and resource controls that it provides. "Companion" protocols make it possible to decompose services into separate protocols that can then be reused in protocol compositions. Capsule propagation can be controlled in a fine-grained manner. "Protocol sessions" provide a flexible adapter between the untrusted mobile code and any end-user applications. In conclusion, we believe that these improvements make great strides toward providing a rich and flexible environment for mobile Java code without sacrificing the security needed by a node administrator.

Availability

Complete Bees source code and documentation are available at `http://www.cs.utah.edu/flux/janos/`.

Acknowledgments

We thank Patrick Tullmann, John Regehr, the anonymous reviewers, and our shepherd, Jonathan M. Smith, for providing many comments and suggestions that helped us to improve this paper.

References

[1] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "PLAN: A packet language for active networks," in *Proc. of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, Baltimore, MD, Sept. 1998, pp. 86–93.

[2] J. T. Moore, M. Hicks, and S. Nettles, "Practical programmable packets," in *Proc. of IEEE INFOCOM 2001*, Anchorage, AK, Apr. 2001, pp. 41–50.

[3] P. Tullmann, M. Hibler, and J. Lepreau, "Janos: A Java-oriented OS for active network nodes," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 501–510, Mar. 2001.

[4] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," in *Proc. of the First IEEE Conf. on Open Architectures and Network Programming (OPENARCH '98)*, San Francisco, CA, Apr. 1998, pp. 117–129.

[5] D. S. Alexander and J. M. Smith, "The architecture of ALIEN," in *Active Networks: First International Working Conference, IWAN '99*, ser. Lecture Notes in Computer Science, S. Covaci, Ed. Springer, June–July 1999, vol. 1653, pp. 1–12.

[6] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith, "A secure active network environment architecture: Realization in SwitchWare," *IEEE Network*, vol. 12, no. 3, pp. 37–45, May/June 1998.

[7] P. B. Menage, "Resource control of untrusted code in an open programmable network," Ph.D. dissertation, University of Cambridge, June 2000.

[8] G. Back, W. C. Hsieh, and J. Lepreau, "Processes in KaffeOS: Isolation, resource management, and sharing in Java," in *Proc. of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, Oct. 2000, pp. 333–346.

[9] Flux Research Group, "JanosVM user's manual and tutorial," July 2002, University of Utah. Part of the JanosVM 0.8.0 software distribution, available at http://www.cs.utah.edu/flux/janos/.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The Flux OSKit: A substrate for OS and language research," in *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St. Malo, France, Oct. 1997, pp. 38–51.

[11] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman, "An OS interface for active routers," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 473–487, Mar. 2001.

[12] P. A. Tullmann, "The Alta operating system," Master's thesis, University of Utah, Dec. 1999.

[13] G. V. Back and W. C. Hsieh, "Drawing the red line in Java," in *Proc. of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, Mar. 1999, pp. 116–121.

[14] H. M. Levy, *Capability Based Computer Systems*. Digital Press, 1984.

[15] Sun Microsystems, Inc., "Java Cryptography Extension (JCE)," http://java.sun.com/products/jce/.

[16] D. J. Wetherall, "Active network vision and reality: lessons from a capsule-based system," in *Proc. of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Kiawah Island, SC, Dec. 1999, pp. 64–79.

[17] J. E. Housley, "FreeBSD HEALTHD daemon software home page," http://healthd.thehousleys.net/.

[18] W. Venema, "TCP WRAPPER: Network monitoring, access control, and booby traps," in *Proc. of the Third UNIX Security Symposium*, Baltimore, MD, Sept. 1992, pp. 85–92.

[19] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec. 2002, pp. 255–270.

[20] Flux Research Group, "ANTS, version 2.0.4," http://www.cs.utah.edu/flux/janos/ants.html.

[21] ——, "Janos Java NodeOS, version 1.2.0," http://www.cs.utah.edu/flux/janos/jnodeos.html.

[22] S. Bhattacharjee, "Active networks: Architectures, composition, and applications," Ph.D. dissertation, Georgia Institute of Technology, July 1999.

[23] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele, and E. Zegura, "Active reliable multicast on CANEs: A case study," in *Proc. of the Fourth IEEE Conf. on Open Architectures and Network Programming (OPENARCH 2001)*, Anchorage, AK, Apr. 2001, pp. 49–60.

[24] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee, "Strong security for active networks," in *Proc. of the Fourth IEEE Conf. on Open Architectures and Network Programming (OPENARCH 2001)*, Anchorage, AK, Apr. 2001, pp. 63–70.

[25] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith, "Safety and security of programmable network infrastrucutres," *IEEE Communications Magazine*, vol. 36, no. 10, pp. 84–92, Oct. 1998.

[26] S. Thibault, C. Consel, and G. Muller, "Safe and efficient active network programming," in *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, West Lafayette, IN, Oct. 1998, pp. 135–143.

[27] J. Waldo and the Jini Technology Team, *The Jini Specifications*, 2nd ed., ser. The Jini Technology Series. Addison-Wesley, 2000.

[28] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler, "The MultiSpace: An evolutionary platform for infrastructural services," in *Proc. of the 1999 USENIX Annual Technical Conf.*, Monterey, CA, June 1999, pp. 157–170.

[29] D. B. Lange and M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, 1st ed. Addison-Wesley, 1998.