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

Tim Stack Eric Eide Jay Lepreau
University of Utah, School of Computing
{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.**

In an attempt to create such an environment, we have developed Bees, an evolution of the ANTS architecture 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, for both users and administrators.

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 received market information, wait for a stock to hit a certain mark, purchase a number of shares, and terminate. Unfortunately, the incredible flexibility afforded by mobile code is offset by the complexity of the "execution environments" that are required to safely execute the code.

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.

US Mail contact address: University of Utah, School of Computing, 50 S. Central Campus Drive, Room 3190, SLC, UT 84112–9205.

Presently, execution environments (EEs) range a great deal in the languages used by mobile code, the lifetime of state on nodes, and the ability to access the contents of a node. Many EEs, including the one described in this paper, rely on the Java language because mobility was part of its original design. Unfortunately, Java will almost always incur some speed penalty, especially in the case of raw packet forwarding in an active network [1]. To avoid this overhead, Java code could be combined with tools more appropriate for forwarding tasks, such as Click [2] or existing TCP/IP protocol stacks. For instance, an active protocol can be separated into a data plane and control plane, where the data plane is a simple Click graph, and the control plane is written in Java.

Besides the choice of language, EEs also vary in the amount of pressure mobile code can exert on a node's resources. Many active network systems, such as PLAN [3] and SNAP [4], restrict the resources consumed by a protocol to those required to forward a single packet. We will 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. But this simplicity limits the possible set of protocols, since there is no way to perform any computation outside 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 the number of packets they process. The "heavyweight" option affords a great deal of flexibility, at the cost of the added complexity required to perform the necessary authentication, protection, accounting, and resource control.

Mobile agent systems use this model in their execution environments, and this model is also the target of our work.

Originally, we used the Active Network Transport System (ANTS) [5] as the EE for our Java-oriented Active Network Operating System (Janos) [6]. The Janos project focused on producing a resource-controlled operating system that could safely execute untrusted Java byte code. Unfortunately, ANTS was envisioned as a lightweight environment, and therefore not always sufficient for protocols that could take advantage of our infrastructure. The lack of authentication and authorization services meant that there was no way for users to acquire additional CPU or network resources, resulting in all network flows having to be treated the same. Furthermore, users could not gain access to simple objects like files or extra classes. Performance was also an issue; for example, ANTS always added an extra layer of routing to any packet transmission. Ultimately, Janos with ANTS simply could not meet the requirements of the applications we wished to deploy.

As a result of this experience, we set a goal to produce a more flexible environment that provided the mechanisms needed to deploy a secure and usable system. Starting with the simple ANTS APIs and programming model, we sought to address the problems we had encountered in authoring ANTS protocols and administering active nodes. The system would address the missing security features by employing a capability-based security model. Since Java is a major component of Janos, its inherent type safety eases 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 would not be built into the core of the system: instead, the cryptographic code and objects would be made available to the mobile code. Simple composition of protocols should be possible so that a primary protocol can make use of the services provided by a "helper." Finally, end user applications should be 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 then provides "services" that an administrator can install to facilitate node discovery, authentication, and authorization.

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. JanosVM

The Janos Virtual Machine (JanosVM) [7] is an Open Source virtual machine for executing Java bytecodes. Unlike any available virtual machine, the JanosVM supports multiple, separate process-like entities (called "teams") within a single VM, without reliance on any underlying OS or hardware support for such separation. The JanosVM supports separate per-team heaps, per-team garbage collection threads, inter-team thread migration, and safe cross-team reference objects. Designed to support asynchronous termination of buggy, uncooperative, or malicious Java bytecode applications, the JanosVM provides robust and scalable multi-process support within a single virtual machine. In the context of Bees, the JanosVM is used to isolate instances of mobile code from one another and control their resource consumption.

B. ANTS

Bees is closely related to the Active Network Transport System (ANTS). 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 capsule reaches a node that does not know how to handle this capsule type, 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, we would argue 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. 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. In contrast, lightweight environments avoid this problem by correlating resource usage to a single packet. Therefore, the resources in use at the node are the same

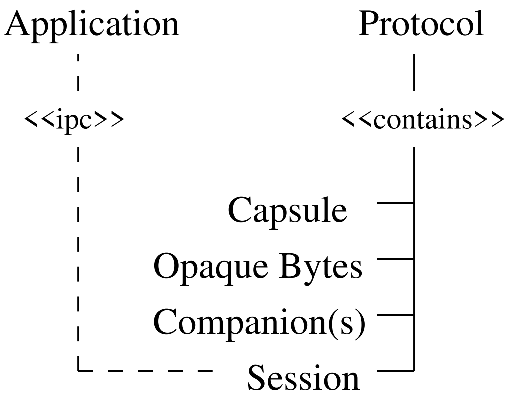

Fig. 1
A BEES PROTOCOL

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 [3, 4]. The alternative is an environment that follows the “heavyweight” model, which is the approach we have taken with Bees.

C. Bees Overview

A Bees-based active node consists of the end-user applications at the top, the “services” that provide basic assistance when interacting with neighboring nodes, the Bees runtime, and finally the JanosVM. The applications are any end-user programs that wish 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.” Next, 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.

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 some service for the main protocol. An example companion would be the system protocol used to download active protocols from neighboring nodes. Next, the protocol can add a “session” class that will act 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 will 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.

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 to 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 completely encoded in the tuple, however, only MD5 hashes of the

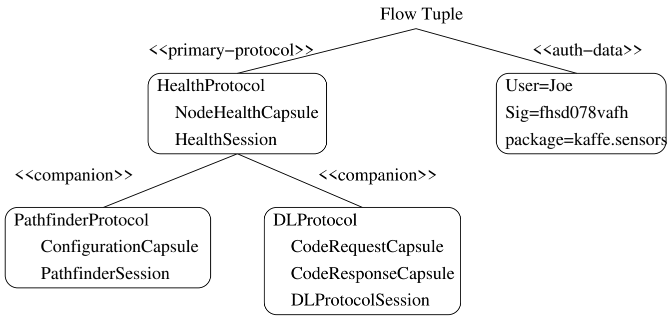

Fig. 2
A FLOW TUPLE

class files or data chunks are 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 grant the protocol access to the `kaffe.sensors` package, if Joe has access to it.

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. 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* [8]. 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 an active protocol is passed a capability to a file by the Auth Agent, 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 and perhaps some cryptographic material. 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. Thus reducing the need to trust the Auth Agent.

Security assurances using cryptographic mechanisms in the Bees runtime is minimized in order to reduce the number of potential vulnerabilities. In case a vulnerability was found, it would be far easier to redeploy the active protocols than retrofit the entire installed base of Bees systems. However, the system does make 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) [9]. 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

Capsules provide a means for neighboring nodes to communicate with one another; however, 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 before deciding where and when to draw the line.

Because Bees follows the "heavyweight" model, the amount of resources consumed to forward a packet does not correlate with the number of capsules forwarded. This flexibility means that mobile code is free 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 the network bandwidth a certain number of times.

ANTS implemented the above semantics through a Time-To-Live (TTL) field in the header of the 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. Then, 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 [10], Wetherall also makes the argument that the TTL is an insufficient means of bounding resource consumption in an active network. For example, a malicious protocol could generate scores of capsules from a single capsule with a TTL of 255. Finally, this method only complicates mobile code that does not follow the typical routing protocol model such as token ring or an agent that periodically sends a capsule.

The Bees implementation maintains the decrement until zero style of the TTL, however, it goes further by supporting more "authorization events" and enumerating the set of operations in which a type of capsule can be used. Unlike the TTL, the limits attached to a capsule are defined in the protocol description by the author and 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 - The capsule is periodically refreshed. 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. The node is then attached to the capsule as the "source neighbor."
- Application Request - The protocol session received a request for some action.

These events then 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.
- forwards - The capsule object is being forwarded to a neighbor node that is different from the original source of the capsule.
- returns - 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 to when implementing a multicast protocol.
- groups - The capsule object is being added to a group of capsules that can then be sent to a neighbor.
- transformTo.capsule_class - 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.

As an example, a ping protocol would attach the "initial" and "returns" permissions with a value of one to its `PingCapsule` type. Now, when the protocol tries to ping a neighbor, the "initial" will be decremented locally, and the "returns" will be decremented when the neighbor sends it back. The above operations are spread across various classes in the system, the most important being `bees.core.Neighbor` and `bees.core.Neighborhood`. These classes are used to send capsules to a directly connected node and a set of neighbors, respectively. The `Neighborhood` is especially useful since a protocol need only set the "neighborhood" limit to the number of times they need to send a capsule, regardless of the number of destination neighbors.

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 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 node'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 `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 protocols are defined by the system or protocol author and called "companions." Companions can then be paired off to provide services for a particular entity.

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, that 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 turn downloading off for a specific neighbor to prevent the protocol from trying to spawn onto that 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 protocol. This part of the pairing process also decides what extra hashes will be used to compute the companion's capsules 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 capsule with the identifier for the mapping request capsule, and the identifier of the unknown capsule. The request is then dispatched to the user flow on the destination node. Finally, this reply is received by the requesting node and used to download more information about the flow. Because the identifiers were split, the system flow was able to receive and evaluate the reply capsule without having to capture packets belonging to a new identifier. In addition, by dispatching the request directly to the user flow, the resources used in processing the request were properly accounted for. However, the most striking part of these events is that two distinct flows were able to communicate, a notion normally denied by the model. 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 routing companion also uses the system downloader since Pathfinder is a user protocol and needs to be transferred along with its own companion.

Besides pairing with a protocol, a companion can also be associated with a flow tuple. In this case the capsules are specialized to the description's hash. Currently, this case is only used by neighbors that wish 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 applications and protocols. Therefore, simply loading the classes 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" will send a GET request to the session object, which causes 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 `write` accepts plain byte arrays that are turned into any number of TCP segments.

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 the 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 sensor information gathered by the motherboard, and report it back to some 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 [11]), and is more substantial than the standard ping and multicast protocols.

A health agents starts its life on the node that wishes to receive the health reports. From there, it uses a companion protocol called "Pathfinder" to find a tree, rooted at this node, that covers all the nodes in the accessible part of the network. Pathfinder accomplishes this by sending out configuration packets that tell the nodes on the local network how many hops from the current node is the root, much like spanning tree protocols. As a byproduct of this process, the configuration packets will cause the other nodes in the network to download the entire agent, resulting in the agent spreading from the root to the leaves of the tree. The use of a companion protocol here relieves the health agent from the burden of determining a route for any of the health report packets. In addition, the separation of 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 flow tuple containing the agent must present the appropriate credentials to the authentication agent on each node. This authentication is done by building a flow tuple requesting the following capabilities:

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

These capabilities will then be passed to the agent when it is first started up 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.

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 [11]. 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.
- 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 tcpwrappers. There is no authentication of requesting entities.

Of course, systems vary in their physical resources. End hosts and edge routers have a large amount of resources that users might want to access, so spending some of them on running a Bees-like system could be beneficial. In contrast, resource-light embedded or specialized machines have already committed the majority of their resources towards one purpose, so they simply have no room or reason to support such an elaborate system as Bees.

C	48255 pps
Java	24882 pps
Java NodeOS	16164 pps
Bees	11018 pps
ANTS	6761 pps

TABLE I

ZERO-BYTE PAYLOAD RELAYING USING UDP

VIII. Performance

We measured Bees performance using the Utah Emulab [12] facility. The test setup was three pc850 nodes, a packet source, the relayer, and the destination. A pc850 node consists of a 850 MHz Pentium III processor running RedHat Linux 7.1 and a recent release of the JanosVM. Table I presents the relay rate for an empty UDP packet or capsule in five configurations. The configurations are: C based, Java Based, a Java relayer that uses the Java NodeOS [13], an ANTS capsule relayer, and a Bees capsule relayer. 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 both ANTS and Bees do not bound what their mobile code can do, they must use this handoff property and experience the same slowdown. 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 GC activity.

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. [5], as the execution environment for mobile code in our Janos active networking project [6]. 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 [14] is on the modular assembly of network services such as active reliable multicast [15]. 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 [16], 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 [9]. 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.

Active networking systems that focus on language-based resource control include PLAN [3], SNAP [4], and PLAN-P [17]. 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. CPU, memory, and network resource controls in Bees are provided with the help of the JanosVM [7], 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 [18].) The JanosVM is derived from KaffeOS [19], developed by Back et al., but tailored for use in an active network setting. The Bees approach is similar to that of the RCANE system by Menage [20], in which resource controls for mobile O'Caml programs are provided by the Nemesis operating system.

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 [21], Gribble et al.'s MultiSpace system [22], and IBM's Aglets for mobile agents [23]. 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" [22] 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, otherwise 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. Finally, we believe that these improvements make great strides towards a rich and flexible environment for mobile code without sacrificing the security needed by a node administrator.

The first release of Bees is scheduled for late December and will be made available at `http://www.cs.utah.edu/flux/janos/`.

References

[1] Erik L. Nygren, Stephen J. Garland, and M. Frans Kaashoek, "PAN: A high-performance active network node supporting multiple mobile code systems," in *Proc. of the Second IEEE Conf. on Open Architectures and Network Programming (OPENARCH '99)*, New York, NY, Mar. 1999, pp. 78–89.

[2] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[3] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott 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.

[4] Jonathan T. Moore, Michael Hicks, and Scott Nettles, "Practical programmable packets," in *Proc. IEEE INFOCOM 2001*, Anchorage, AK, Apr. 2001, pp. 41–50.

[5] David J. Wetherall, John V. Guttag, and David 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.

[6] Patrick Tullmann, Mike Hibler, and Jay 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.

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

[8] Henry M. Levy, *Capability Based Computer Systems*, Digital Press, 1984.

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

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

[11] James E. Housley, "FreeBSD HEALTHD daemon software home page," `http://healthd.thehousleys.net/`.

[12] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[13] Flux Research Group, "Janos Java NodeOS, version 1.2.0," `http://www.cs.utah.edu/flux/janos/jnodeos.html`.

[14] Samrat Bhattacharjee, *Active Networks: Architectures, Composition, and Applications*, Ph.D. thesis, Georgia Institute of Technology, July 1999.

[15] 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.

[16] 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.

[17] Scott Thibault, Charles Consel, and Gilles 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.

[18] Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman, "An OS interface for active routers," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 473–487, Mar. 2001.

[19] Godmar Back, Wilson C. Hsieh, and Jay Lepreau, "Processes in KaffeOS: Isolation, resource management, and sharing in Java," in *Proc. of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, Oct. 2000, USENIX Association, pp. 333–346.

[20] Paul Menage, "RCANE: A resource controlled framework for active network services," in *Active Networks: First International Working Conference, IWAN '99*, Stefan Covaci, Ed., vol. 1653 of *Lecture Notes in Computer Science*, pp. 25–36. Springer, June–July 1999.

[21] Jim Waldo and the Jini Technology Team, *The Jini Specifications*, The Jini Technology Series. Addison-Wesley, second edition, 2000.

[22] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David 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.

[23] Danny B. Lange and Mitsuru Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, first edition, 1998.