Bees User's Manual and Tutorial
Version 0.5.0

Flux Research Group
School of Computing
University of Utah

http://www.cs.utah.edu/flux/

December 24, 2002

Contents

Chapter 1

Introduction

Bees is an active network Execution Environment (EE) that downloads and executes Java code. Originally based on the ANTS environment, Bees seeks to provide stronger guarantees to the node administrator and a richer environment for mobile code authors.

The goal of this document is to describe the things you need to know in order to start using and developing for Bees, including:

- An overview of the system and its possible uses (Chapter 2).
- How to install Bees on your system (Chapter 3).
- A tutorial on how to use Bees (Chapter 4).
- The high level design of the system (Chapter 5).
- Instructions on interacting (Chapter 6) with a Bees node.

Chapter 2

Overview of Bees

This chapter presents a high level overview of the concepts and reasoning behind much of Bees. You should be able to skip directly to the installation and tutorial if you are in a hurry, however, some things will be hard to fully understand without this background.

2.1 Motivation

A typical computer has scores of data and functionality that is, often times, hard to reach through a network. Presently, a machine makes itself network accessible by running a server that communicates with clients through an arbitrary set of protocols. While this system works, it tends to create a strong binding between the data, the protocol, the software, and the administrator. In other words, the data is “only” available through a single protocol, whose functionality is bounded by the latest implementation installed by the machine’s administrator. Alternatively, an “active” system is based on mobile code that allows clients to define the remote processing and protocols used to communicate, while still allowing the administrator to set the terms of the interaction with the node. However, mobile code cannot do anything useful without a rich environment already existing on the network nodes. Nor will an administrator execute an environment that did not provide strict controls over the resources consumed and accessed by the mobile code.

Active Networks research has addressed many of these issues. For example, the ANTS EE provided an environment for downloading and executing Java code. A “protocol” could be written that communicated with other network nodes using active network packets called, “capsules.” Then, whenever a node received an unknown capsule type, it would download the protocol and install it on the node. Once installed, the protocol could perform whatever processing it wished on any received capsules. However, the amount of processing possible was severely limited because the class name space was thinned to prevent mobile code from accessing local resources (e.g. files, threads, etc...). In addition, mobile code had no guarantee on the amount of time node local state would survive. If more protocols and data were loaded into the node, older state would be pushed out. Nonetheless, ANTS was used extensively in the community, likely because of the properties of the Java language.

In an effort to remedy the shortcomings of ANTS and to further explore areas of active networks, we have implemented Bees. Because of the success of ANTS, Bees shares much of the same overall design. Mobile code authors develop capsules that are contained within protocols that are then deployed into the network. Beyond this point though, Bees begins to diverge. Node administrators can now take advantage of a flexible authentication and authorization mechanism, making it possible to reject some protocols and grant extra privileges to others. Capabilities provide the means for downloaded code to access privileged resources, such as files and neighboring nodes. There is basic system support for composing protocols and interacting with system support protocols. Finally, there are a large number of more minor improvements that should make the system easier to program for and use.

2.2 Use Cases

This section will covers the use cases that shaped the design and implementation of Bees. There are three cases, each designated by their relative location in the network.

2.2.1 Active Route-point

An Active Route-point is a router that is willing to run Bees. The major advantage of this case is the ability to "boost" legacy protocols into a downloaded protocol. In addition, a route-point is the optimal place to install multicast protocols since it is trivial for such a node to send a single packet through multiple network interfaces. The down sides of this scenario is that the router may not have the spare processing power to support Bees, relegating its use to routers near the edge of the network.

An example using Bees in this form might be an intrusion detection system that monitored flows between the "inside" and "outside". The system could install some basic, low level monitors on each network flow that reported "suspicious" activity to the application. Once a suspicious flow was found it could be boosted into the slower, but more flexible Bees environment, for further analysis.

2.2.2 Active Midpoint

An Active Midpoint is a well connected node whose sole purpose in life might be to run Bees. The objective of this scenario is to provide an intermediate point in the network where packets from other nodes can rendezvous and be fused, processed, or redistributed, with the ultimate goal being reduced load on the network. It differs from the routing scenario in that the protocols require the cooperation of the end points, packets cannot simply be captured and boosted as they pass through the router. For example, a Squid web cache network receives requests from end points and satisfies them from its cache, its peer's caches, or the original source. Now, with Bees we are able to make a node's favorable location available to a large number of protocols without relying on the administrator to install a protocol processor.

An example using Bees in this form might be a network game that performed fusion and pruning between end points and peer midpoints. On startup, the game could locate an existing or friendly midpoint on which to install itself. Then, the end points would transmit their changing game state to the midpoints where they could decide what state needed to be forwarded. For example, player state can be pruned at a midpoint for those that, in the game, are far away from the other players.

2.2.3 Active Endpoint

An Active Endpoint is a local node with a single network interface that is executing Bees. The primary objective of this scenario is to make the data and functionality of the node accessible through the network without locking in the protocols used to access this data. For example, there are current protocols like SNMP and Healthd [2] that are used for accessing management related data over the network. Unfortunately, these separate protocols split the accessibility of data, provide little to no remote processing, and are hard to change in the face of new security threats. Alternatively, an end point executing Bees could easily make all the data available to protocols, allow downloaded code to post process data before transmission over the network, and make it possible to implement and deploy a protocol that implements new security counter measures.

Following the previous management example we could imagine a protocol that injected itself into every node in a specific administrative domain. Depending on the policy, this protocol could monitor CPU temperature, perform periodic diagnostic tests, and then report anomalous results to the administrator. The ability to test the results on the node without having to transfer them back to some central testing node could greatly reduce traffic and load, especially if the tests require a great deal of historical data.

Chapter 3

Compiling and Installing Bees

The `INSTALL` file in the topmost directory of the Bees source tree contains an abbreviated version of these instructions.

Bees consists of the following components:

`beesvm`	A JVM wrapper script that adds the Bees classes to the class path.
`bees`	A short script that starts up a "blank" Bees node.
`nodesetup`	A script that starts up one or more instances of JVMs executing Bees, configures them, and starts up one of the `apps.authenticator` applications.
`pylib/*`	A collection of `python` scripts that assist in communicating with Bees nodes.
`webmin/*`	A collection of web pages and CGI scripts that assist in interacting with Bees nodes.

3.1 Acquiring Bees

Bees is available on the World Wide Web under `http://www.cs.utah.edu/flux/janos/bees.html`. Currently, Bees is distributed only as source code.

3.2 Building the Bees Programs

In order to build Bees from source, you will need:

`Java compiler`	We use `jikes` v1.14, but any compiler should work.
`Java Virtual Machine`	Currently, we have only tested Bees with JDK 1.4.1 and the latest JanosVM. In the case of the JanosVM, Bees will also take advantage of some of its extensions, such as the `kaffe.sensors` package.
`Java Cryptographic Extension`	JDK 1.4 already comes bundled with the cryptography classes. The JanosVM requires a separate implementation, such as the one available from `http://www.bouncycastle.org`.

`jar or zip`	These are needed to produce the archives that contain the compiled classes.
`python`	The `python` language is used extensively to interact with Bees nodes. We use version 2.2.
`latex`	This program is optional, however, you will need it to rebuild the TEX based documentation.
`javadoc`	Again, this program is optional, however, you will need it to rebuild the Java class documentation.
GNU `make`	We make use of some `gmake` extensions so regular versions won't work. We use GNU make 3.79.1.
Disk Space	XXX Fix me

Once you have the above-listed tools, you will need to (1) unpack the Bees source distribution `tar` file and (2) create a build tree to contain the object files. Assuming that you did these things in your home directory, your directory setup would look something like this:

`~/bees-0.5.0/`	The Bees source tree, unpacked from `bees-0.5.0.src.tar.gz`.
`~/obj/`	The directory you create to be the root of your Bees object tree. (Of course, you can call this directory anything you like. It does not have to be called `obj`.)

3.2.1 Configuring Bees

Given the directory setup described in Section 3.2, you would type the following commands to configure Bees for your operating system:

```
cd ~/obj
../bees-0.5.0/configure
```

The `configure` script in the root of the Bees source tree is an ordinary `configure` script, generated by GNU `autoconf`. Type "`configure --help`" to see all of the script's command line options. In addition to all of the standard options accepted by `autoconf`-generated scripts, Bees's `configure` script accepts the following Bees-specific options:

`--enable-optimized`	Use the optimized versions of JAR files when possible.
`--with-sensors`	Specify a different Sensors jar file to compile against. A copy of the `sensors.jar` from the JanosVM is already included in the `lib` directory.
`--with-jnodeos`	Specify a different Java NodeOS jar file to link against. This option defaults to the JAR file included in the `lib` subdirectory of the distribution.
`--with-jvmulation`	Specify a different JanosVM emulation library to link against. This option defaults to the JAR file included in the `lib` subdirectory of the distribution.

`--with-jvmkit`	Specify a different JanosVM kit library to link against. This option defaults to the JAR file included in the `lib` subdirectory of the distribution.
`--with-classpath`	Specify a different class path to compile against. The default is to use the one assumed by the compiler, so, if you're using `jikes`, you'll need to use this option.
`--with-link-classpath`	Specify a different class path to use in the JVM wrapper scripts. Defaults to the above compile class path.

Besides the command line arguments the configure script will also pay attention to the following environment variables, corresponding to the various programs used to build and execute Bees: JAVA, JANOSVM, JIKES, JAVAC, JAR, ZIP, JAVADOC, PYTHON, LATEX.

3.2.2 Compiling Bees

Finally, you are ready to compile Bees! Simply type the following commands to build Bees and its auxiliary programs:

```
cd ~/obj
make all
```

Assuming all goes well, these commands should produce all of the Bees programs that need to be compiled. In addition, if you wish to rebuild the documentation you can use `make` with the "documentation" target.

3.2.3 Checking and Installing Bees

Once the system has been compiled you'll want to run the test suite to make sure everything is working properly:

```
cd ~/obj
make check
```

Finally, if everything is working well enough you can install the system in the `\snna{}` directory by executing:

```
cd ~/obj
make install
```

Now, you'll probably want to try out the tutorial in the following sections.

Chapter 4

Tutorial

This chapter takes you through the steps needed to run some simple example programs. Complete source code for each of the examples is located in the `src/apps` subdirectory of the Bees software distribution.

4.1 Health Daemon

The first part of the tutorial will demonstrate the use of the included Health protocol. This protocol spawns onto all of the nodes it has access to, polls the motherboard sensors, and reports the digested results back to a central server. The source for the protocol is in the `src/apps/health` directory and the source for the helper protocol, Pathfinder, is in the `src/apps/pathfinder` directory.

4.1.1 Health Overview

The Health protocol is designed to be started on a "server" node that is willing to receive health reports from any number of clients. From there, it uses a helper protocol called "Pathfinder" to find a tree, rooted at the server, 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 helper 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. Once, the nodes have figured out the path to the root of the tree, they will begin to poll their motherboard sensors and generate health reports. These reports are then signed with the node's private key and forwarded to their root neighbor, until they reach the server node where the report is stored for later retrieval by the user.

4.1.2 Getting Started

If you have access to the Utah Netbed [1] we provide a late-model Linux OS image, RHL-WITHBEES, that contains all the necessary components for this tutorial. The location of the Bees files in the image are as follows:

`/usr/local/bees`	The Bees installation directory.
`/local/bees`	A copy of the Bees source.

`/local/bbuild`	The build directory for `/local/bees`. Simply type "make" and "sudo make install" if you make any changes that you wish to be reflected in the installed version.

If you do not have access to the Netbed, you will need the following in addition to Bees:

- A Linux machine with `libsensors` installed or a FreeBSD and the `xmbmon` [3] program.
- The latest version of the JanosVM.

4.1.3 Begin Netbed Experiment

If you're using the Netbed, create an experiment with the following ns file, or something like it:

```
set ns [new Simulator]

# Load testbed specific stuff
source tb_compat.tcl

# Create the nodes and set their OS types.
set router [$ns node]
tb-set-node-os $router RHL-WITHBEES
set client1 [$ns node]
tb-set-node-os $client1 RHL-WITHBEES
set client2 [$ns node]
tb-set-node-os $client2 RHL-WITHBEES
set client3 [$ns node]
tb-set-node-os $client3 RHL-WITHBEES
set server [$ns node]
tb-set-node-os $server RHL-WITHBEES

# Link the router and the server.
set link0 [$ns duplex-link $router $server 100Mb 0ms DropTail]

# Build the lan for the router and clients.
set lan0 [$ns make-lan "$client2 $client1 $client3 $router" 100Mb 0ms]

# Connect the router and clients to the lan
tb-set-node-lan-params $client2 $lan0 0ms 100Mb 0.0
tb-set-node-lan-params $client1 $lan0 0ms 100Mb 0.0
tb-set-node-lan-params $client3 $lan0 0ms 100Mb 0.0
tb-set-node-lan-params $router $lan0 0ms 100Mb 0.0

$ns rtproto Static
$ns run
```

Netbed should then create a network consisting of five computers, three clients that will be reporting health information, a router that simply forwards capsules, and a server that gathers all the information.

4.1.4 Starting the Nodes

Initializing a node consists of two phases, starting Bees in a JVM, and configuring the node through a separate interface. The quickest way to do both is with the included `nodesetup` script installed in `/usr/local/bees/bin`. The script needs to be told what interfaces to use based on the "IP0, IP1, ..., IPN" environment variables. For example, to start up a Bees node that works over two interfaces, you would run `nodesetup` like so:

```
gir:~> env IP0=192.168.1.2 IP1=192.168.2.2 /usr/local/bees/bin/nodesetup
```

However, in the case of a Netbed node, the script will automatically figure out what network interfaces it should use to discover nodes on the same LAN segment. So, just `cd` into a directory, preferably one in NFS, and execute the following to start a node on the Netbed server node:

```
server:~/bshared> sudo env /usr/local/bees/bin/nodesetup
...
BDomain(57) - Notice: Discovery Log:Sending advertisement
BDomain(57) - Notice: Discovery Log:Received own advertisement.
BDomain(55) - Notice: Liveness Log:Sending poke.
BDomain(55) - Notice: Liveness Log:Sending poke.
```

Since this is the first time you have started a node on this machine it will take a little while to boot up because it needs to generate new DSA and RSA keys. To speed up subsequent startups, the keys will be written to a file in the current directory. After this, the node will use the Discovery protocol to multicast advertisements of itself and listen for advertisements from neighbors on the same LAN segment. Of course, there are no other nodes to find, so login to the other machines, `cd` to the same shared directory, and boot Bees. If all went well, you should see the occasional message about neighbors being found and the Liveness protocol reporting round trip times between nodes:

```
BDomain(520) - Information: Auth App Log:Found neighbor: ...
...
BDomain(528) - Notice: Liveness Log:Neighbor: TypeID[hash=byte[length=16; data={
10ccf498e78ec25279cd606a5c351fbe }]]; Elapsed Time: 1ms
```

Unfortunately, we do not have a key distribution protocol, so you will need to kill and restart all the nodes so they can find each other's key files. If you cannot start the nodes in an NFS directory you will need to manually copy the key files to each node or grab a floppy disk and a pair of sneakers.

4.1.5 Node Control

After the nodes have finished rebooting, you can access the web based administration interface by pointing a web browser at `http://server.<eid>.<pid>.emulab.net:26000`. These pages allow you to create objects, start protocols, and perform other simple maintenance tasks. Browse around a bit to familiarize yourself with the layout of things, if you need help, simply click on the "Help" link in the menu bar on the left.

One of the major components of Bees is a "capability", an object that grants certain privileges to those entities which hold a reference to it. By clicking on the "Capabilities" link in the menu bar you can browse the tree of objects currently in the kernel. In addition, you can create new ones by clicking on the "Create" button of a directory and filling out the form.

Lets create a new file capability that controls access to the file “foobar” in the node’s current working directory. First, click on the “Capabilities” link in the menu bar to go to the root of the capability tree. Next, click on the “Create” link at the far right of the “Current Directory” listing. This should bring up a form containing some text fields and a menu, select “File” in the menu to indicate that you want to create a file object. Now, fill out the “Name” field with the name you would like to give the capability. This name, prefixed with the destination directory, can then be used to reference the capability in the future. Finally, type “foobar” into the “Path” field and hit the “Create” button. If all went well, the page should report success, and you should be able to reload the “Capabilities” page to see the new object. There are a number of other types of capabilities that can be created, check out the help file for more information. Also, consult the administration chapter (6) for more details on `nodesetup` and the webmin interface.

4.1.6 Building a Health Flow Description

Now that you have some experience with the webmin interface, we can start setting up the Health protocol. The nodes you’ve started are automatically configured to accept protocol’s from a known source and reject any others. Therefore, we must produce a description of the Health protocol signed with a key pair known to us and the other nodes in the network. Then, when the protocol tries to spawn on to a node it can present the signed description to the node’s authentication and authorization application.

Constructing a description is done with the “Flow Builder” application on the “Tools” page, reachable by the link in the menu bar. Initially, the form will ask for a JAR file containing partial flow descriptions that can be turned into a real description. The default value points to the installed `apps.jar` file, which contains the Health protocol, so just hit the refresh button. Since there can be more than one description in a JAR file, the page will display a menu containing the ones it has found, so select “Health Demo” and hit the refresh button again. Now, you should be looking at a new form with a short description of the protocol and the list of capabilities that the flow would like to be given access to. Most of the fields should have reasonable defaults, you only need to fill out the “Destination file capability” with the name of a file capability that will be used to store the manufactured description. You can use the prebuilt one in `/users/demo/health/flow.desc` or the one you created in the previous section. After you have filled out the field, hit the “Build” button at the bottom of the page to create the flow description. Most likely, the build should work, however, the only way to know for sure is to look for exceptions on the console.

The other parts of the form are used to communicate some data to the authenticator application running on a node. In this case, the “source-auth.source” parameter tells the “Source Based” authenticator that our user name is “demo”. Next, the “source-auth.algorithm” parameter indicates that the description is cryptographically signed using the DSA algorithm. Finally, there is a list of the capabilities that we wish the flow to have when it is started on the node. The list contains the source of the capability, a path relative to the user directory, any manipulations to the capability’s permissions, and an optional destination name that the protocol can use when looking up the capability within its own domain. For the Health flow, the capabilities are as follows:

`cpuMeter` The pool of CPU time that this flow will pull from.

`memMeter` The pool of memory that this flow will pull from.

`classes` The set of classes to be passed to the node. The default is the `classpath/sensors.jar` directory in the “demo” user’s directory. Passing a directory to a domain will result in all of its children being added to the domain. So, this will give the protocol access to the “kaffe.sensors” classes.

`packages` Similar to “classes”.

`nodePrivateKey`	The node's private key, used to sign health reports. This address is prefixed with "global" to denote a global reference instead of one relative to a user directory. This capability also takes advantage of the "bound permissions" option to ensure that the private key can only be used for signing data and not decryption.
`nodeKeys`	A directory containing the public keys for the other nodes in the network.

4.1.7 Starting the Health Flow

With a valid flow description you can start a protocol that will be able to spawn onto all of the nodes in the test network. You can do this by using the "Protocol Launcher" application in the "Tools" page. Again, just hit "Refresh" to use the installed `apps.jar` and then select "Health" from the option menu. The form should now have several empty fields that need to be filled out before proceeding. The "Flow description file" should be set to the file given to the builder in the previous section (e.g. `/users/demo/health/flow.desc`). Next, the "Resource Meters" field is used to tell the system the CPU and memory resource pools to pull from. For this example, just enter "/cpu-meter, /mem-meter" and leave the default values in the "CPU Shares", "CPU Threads", and "Memory Pages" fields. Next, there a list of capabilities that will be passed to the flow on this node. The capabilities, and their suggested values, are as follows:

`logger`	Select the "Debug" level. The "logger" capability is used by the protocol to log messages to the console, you can adjust the message severity level with the option menu. Also, you can select the custom level and enter the name of a logger capability to use instead of the auto generated one.
`Extra Capabilities`	Prefix the `/classpath/sensors.jar` value with `/users/demo`. This area is used to pass any extra capabilities you wish the flow to have. Typically, it is used for passing classes and packages since the system will automatically add them to the domain's class loader.
`user/clock`	Use `/clock`, the primary system clock.
`user/nodeKeys`	Use `/nodeInfo`, the directory of node keys.
`user/serverKeyPair`	Use `/users/demo/health/keys/DSA`, the key pair the server should use when sending requests to nodes.
`user/nodePrivateKey`	Use `/nodeKeys/DSA/private`, the node's private key.

Now, hit the "Start" button and you should end up at the "health monitoring" page. This page will talk to the Health protocol on the local node and display any reports received from the client nodes. Because it is so early in the protocol's lifetime, the page will only display "No clients yet." However, this will change as the protocol is transferred and installed on the other nodes. Therefore, you will want to reload the page to get an up-to-date view of what the client nodes are reporting.

Terminating the Health agent can be done by clicking on the terminate button in the health monitoring page or the status page. The agent is designed to terminate from the inside out. In other words, terminating the protocol on the server node will cause the other nodes in the network to timeout and self-terminate. This self-destruct mechanism works because the protocol is designed to not respawn onto nodes that make up the path to the server node. Alternatively, the agent *will* respawn onto any client nodes that are restarted since they will be the target of the Pathfinder configuration capsules.

Chapter 5

Design

This chapter will give a high level overview of the major components in the Bees design. For a more detail, consult the implementation chapter and Javadoc.

`Protocol`	The specification of packets, processing, and data used to communicate with other nodes and applications.
`Protocol Flow`	The specification of the protocol, any flow helper protocols, authorization, and other data needed to define a type of network flow.
`Domain`	The process-like abstraction that isolates a protocol flow.
`Capability`	A privileged object managed by the system.
`Neighbor and Neighborhood`	A directly connected node or set of nodes that a protocol can infect.

5.1 Protocol

A protocol in Bees defines how nodes in the network will communicate with each other and any applications. This specification is made up of three components, the *Capsules* which define packet types, the *Protocol Session* which defines the method used to interact with the application, and any additional code or data needed by the other two.

5.1.1 Capsule

A capsule defines the wire format of packets sent to other nodes and the expected processing of those packets. On the wire, a capsule is made up of the following:

- The transport protocol header, any transport could conceivably be used, from Ethernet to UDP.
- The Bees header version number.
- A set of flags, currently unused.
- A per-capsule unique identifier that determines what code and context should process this capsule. The value for this identifier is an MD5 hash covering the contents of the protocol flow and the capsule's byte code or a version string.

- An optional per-capsule unique identifier that is used to further specialize what code and context should process the capsule. This extra identifier is used to distinguish between domains that may be using the same system protocol (e.g. the protocol downloader).
- And finally, the user payload.

Unlike the original Bees, there are no addresses or implicit routing implemented by Bees. Instead the user protocol is expected to perform these tasks directly or make use of other protocols in the system. For example, a protocol interested only in executing on the end nodes can use a built-in protocol like UDP or a user routing protocol for transporting capsules between the end hosts. Additionally, this allows Bees to restrict the set of neighbors accessible to a protocol because the destination is encapsulated by an unforgeable object instead of just a random integer.

Again, unlike the original Bees, the system does not enforce a time-to-live field for every capsule. The alternative is to restrict capsule propagation using a specification given when the protocol is defined. Some example forms of propagation include, returning to the source node, forwarding to another node, and forwarding to a group of neighbors. This method allows Bees to exert tighter control within the node, without arbitrarily restricting some forms of protocols. For example, a proposed anonymous protocol behaved like a token ring network, a single capsule would loop through the network carrying data that was added/removed along the way. While, this could conceivably be implemented in the original Bees, it would involve tricking the system a bit, which only emphasizes the lack of control provided by the TTL.

Of course, without a system maintained TTL, some may fear that a routing loop in the user protocol could cause a great deal of damage to the network. However, these fears seem driven by thoughts of a routing loop in an IP network, which does not directly translate to a Bees network. First, the majority of user protocols are not expected to act as a medium for other protocols, so a loop in one will not affect the others. This is unlike IP, where a loop will disable UDP, TCP, ICMP, and several other protocols. Second, the resource controls provided by the lower layers of the system will guarantee resources for peer protocols and most likely force a bottleneck at a node in the network. Finally, the ability to kill any protocol in the network leaves the door open for more sophisticated methods of controlling rampant network flows. Taken together, these differences reduce the utility of the TTL to the point where it is more of a burden on the protocol writer than a safety mechanism for the network.

Once a capsule has reached a node in the network it is delivered to the protocol code for unmarshalling and evaluation. The unmarshalling process will typically cover exposed data that is needed for the evaluator to do the minimum amount of processing. Any additional data that needs to be manipulated with cryptographic operations is typically left for the evaluation method to process. For example, a capsule carrying an encrypted payload might want to delay decryption until it reaches the destination node containing the appropriate keys. Finally, the evaluation method is called to perform node local operations: update capsule data, forward to peers, and/or send packets to interested applications connected to the node.

5.1.2 Protocol Session

A "protocol session" is an intermediary that translates application requests into protocol actions and, in turn, informs the application upon completion of those actions. For example, the session for a code downloading protocol would present applications with an interface for starting the transfer of a set of code to the local node. Notice that we do not present the protocol to the application in the form of its capsules, this is a completely different conversation than the one done between protocols on peer nodes. While this means that a second protocol must be established, the inconvenience is outweighed by the ability to insulate the application from changes in the internal protocol. In addition, this external protocol makes it possible for applications to interact with existing legacy protocols and active protocols that they did not originate. For example, an older application can download and use a newer version of a protocol, assuming the external protocol did not change. Finally, the explicit nature of the session forces the protocol writer to firmly define what the protocol is responsible for doing and what the application is responsible for doing.

5.1.3 Protocol Extras

5.2 Protocol Flow

The *Protocol Flow* in Bees is used to wrap a protocol's communication methodology with user specific authorizations and data. This extra data is required because instantiating a protocol on a node is not always as simple as downloading and installing the code. For instance, some administrators might not want anonymous code to run on their node or the code might need access to some privileged operations on the node. In order to address these requirements without mandating too many things, Bees allows users to couple identifier/value pairs onto the description of a protocol using a `ProtocolFlow` object. This description can then be automatically examined by authentication code running on the node to determine the amount of access to grant new or existing protocols. Additionally, this makes it possible to parameterize protocols sent into the network.

For example, the `apps.health.HealthProtocol` requires access to the motherboard sensors in order to report the health of a node. However, the node's administrator may not want to give access to the sensors unless a protocol was signed with a specific key. Therefore a flow description must be built containing the identifiers of the protocols, capsules, data, a request for access to the sensors and a signature covering these bytes. Finally, when the protocol attempts to spawn onto a node, this description will be passed to an authenticator application, which will verify the signature and start the flow with the appropriate privileges.

5.3 Domain

Protecting and isolating protocols, applications, and the kernel in Bees is the job of the *Domain* construct. Its an important component since the execution of untrusted code requires a great deal of care to ensure that it cannot damage other protocols by changing data, starving them of resources, or gaining access to privileged objects. Fortunately, many of these issues are taken care of by lower layers, such as the JanosVM and NodeOS. The responsibility left to Bees and its users is deciding when to start up and stop a domain. However, we can allow the protocol to determine when a domain is terminated, for example, if it hasn't sent or received a packet after a specific amount of time. In the end though, protocol domains are always at the mercy of any administrator applications.

5.4 Capability

A *Capability* is used to tightly control access to objects outside of a protocol's domain. In Bees a capability is made up of a per-node unique identifier, one or more java classes, a set of permissions, and a flattened version of the java objects. The purpose of the identifier is to allow disjoint domains to address the same capability and to prevent spoofing. The java class serves to provide the actual implementation and restrict what actions untrusted code can perform. The permissions allow the user/administrator to exert finer grained control over the operations allowed by classes. And finally, the flattened versions of the java objects, the capability material, allows for persistence and a private communication channel between domains. An example of how these components relate to a symmetric cryptographic key capability might look like this:

- The unique identifier allows a session key management protocol to pass the capability to protocols interested in encrypting their packets.
- The java classes can be used to encrypt and decrypt packets in the session.
- The permission set can have a permission for calling the `getEncoded` method, making it possible to deny foreign domains access to the key material, without stopping them from using the key.
- The capability material consists of the data that needs to be protected from the user code, in this case, the algorithm name, extra parameters, and the secret key material.

Unlike previous versions of Bees, capabilities play the major role in the security of the system, there are no principals or access control lists. Delegation of capabilities is the role of the administrator when directly manipulating the kernel or an authorizer application which makes its decisions based on the protocol flow description.

5.4.1 Capability Envelopes

Transferring capabilities between domains is done by first wrapping it in an envelope, manipulating the permission set, and passing it to the domain over a communication channel. The domain can then unseal the envelope to retrieve the capability, possibly with a diminished set of permissions.

5.5 Neighbor and Neighborhood

A *Neighbor* in Bees is a directly connected node that a protocol can infect. And, a *Neighborhood* is a set of neighbors grouped together in order to make it easier to account for multicasting and other redundant uses of capsules. Some examples of neighbors are:

- Nodes on the same Ethernet LAN.
- Nodes reachable through UDP.
- Nodes reachable through UDP multicast. Useful for doing a less intrusive broadcast on a LAN.

Again, unlike Bees, Bees will not do anything unless absolutely necessary, so it will not route packets to neighbors not directly addressable by the transport. Also, Bees will not automatically listen for capsules from all neighbors given to a domain, the protocol must manually attach the listeners. This may seem unpleasant, but it allows protocols to avoid having to listen to neighbors they create, useful for applications designed to discover neighbors.

5.6 Companion Protocols

A network protocol can rarely perform its task by itself, typically there are one or more protocols working in the periphery to assist the primary protocol. For example, TCP runs on top of IP, which in turn can interact with DHCP, ARP, and a number of routing protocols. Similarly, Bees protocols need to make use of other protocols to transfer themselves to neighbors or perform basic routing. To support this interaction, Bees allows a protocol to attach "companion" protocols to themselves. These helper protocols are then included as part of the protocol flow and instantiated alongside the primary protocol. For example, the included Health protocol uses the Pathfinder protocol to perform simple routing. When this flow is instantiated the

Pathfinder protocol executes a spanning tree like algorithm to find routes between a root node, determined by the Health protocol, and a number of clients. Next, the Health protocol uses the information gathered by Pathfinder to route its report capsules back to the root node.

Implementing a companion is done by adding one of the `CompanionProtocol` sub-interfaces to the `Protocol` description. XXX complete me

These protocols are very similar to user defined protocols with the exception that their capsules make use of the "specialized" capsule identifier. This specialized identifier is used to differentiate capsules between multiple domains. For example, a code downloader capsule has its own unique identifier and a specialized identifier corresponding to the class to be downloaded.

5.6.1 Built-in Companions

This section will describe the system provided companion protocols that support user protocols in Bees. Unlike a typical protocol, some of these companions can cross flow boundaries. For example, a flow that has yet to be downloaded to a neighbor will need to communicate with one of these system flows before it can be transferred over.

Protocol Downloader

The protocol downloader is used to reliably transfer a protocol's byte code and data between directly connected nodes. This protocol's description is built by the `bees.core.DLProtocol` class and should be added as a companion to any protocol that wishes to be downloadable. The Protocol consists of three classes, two capsule types used to transfer byte arrays between nodes, and the ProtocolSession which manages any in-progress downloads. New protocols are downloaded by first calling one of the `register` methods on the current `Node` object. This method will use a `DLProtocolPresentation` to start and control the download. Once the download successfully completes, the presentation's `downloadComplete` method will be called. At this point the protocol's byte arrays have been completely downloaded, the only thing left is to boot the protocol and bind a `ProtocolPresentation` to the new flow.

ID Translator

The ID translator is used to map unknown capsule identifiers to protocol identifiers. This protocol's description is built by the `bees.core.IDTranslationProtocol` class and should be added as a companion to any protocol that wishes to be downloadable. The protocol is designed to be registered by the authenticator application and then used to detect new flow identifiers that can be passed along to the flow downloader.

Flow Downloader

The flow downloader is used to transfer the description of a protocol flow between directly connected nodes. This protocol's description is built by the `bees.core.UnknownProtocol` class and should be added as a companion to any protocol that wishes to be downloadable. Typically, this protocol is started in its own flow by the authenticator application to download descriptions for a given flow identifier. Basically, the sequence of events is:

- An unknown capsule type is received from a neighbor.
- The `IDTranslationProtocol` sends a capsule back to the neighbor requesting a mapping from the unknown capsule identifier to a flow identifier.

- The neighbor replies to the request with the flow identifier which eventually gets sent to the authenticator application.
- The authenticator makes a decision whether or not to download the description for this flow identifier.

Chapter 6

Administering Bees nodes

This chapter will describe how to setup and control Bees nodes. Initially, a node starts out “blank” and needs to be told what to do via a scripting interface. Once, the node’s capabilities have been loaded or created, the authenticator application can be started and the node is free to accept new protocols.

Note: Unfortunately, some functionality is still missing in this version of Bees.

6.1 Node components

This section will cover the key components in an Bees node.

6.1.1 Default Capabilities

For a node to function correctly it needs several sets of capabilities to be created or loaded into the system. These capabilities will then be passed to the applications and protocols that are to be started on the node. Initially, a node has a set of capabilities that correspond to the built-in components of the machine. For instance, there is always a console object for interacting with standard I/O and a capability to access the current working directory.

6.1.2 Authenticator

The authenticator application is the main application on a node, it takes care of verifying new protocol flows, starting them up, and passing them capabilities. Currently, there are only two authenticators in the `apps.authenticator`, Everything and SourceBased. The former allows any flow into the system with some minimal set of capabilities. And, the latter checks a signature in the flow description against the source’s key and passes it whatever capabilities it requested. Besides verifying protocol flows, the authenticator applications also locate a node’s neighbors by starting and interact with the Discovery and Liveness protocols.

6.1.3 Discovery and Liveness

Locating and probing neighbor nodes is the job of the Discovery and Liveness protocols found in the `apps` package. The Discovery protocol is used to multicast an advertisement containing information about the node and to receive these advertisements from other neighbors. Once an advertisement has been received, it is passed to the authenticator for further handling. The authenticator then uses the data in the advertisement to construct a new neighbor capability which can be passed to the domains under its control. One of these domains contains the Liveness protocol, which monitors the neighbors to ensure the reachability. If Liveness finds one of the neighbors to be unresponsive it informs the authenticator so that the neighbor can be removed from the other domains.

6.2 Scripts

A number of node startup scripts have been provided to assist in getting a node setup.

6.2.1 beesvm

The `beesvm` script is a simple wrapper that appends the Bees JAR files and dependencies onto the class path. Use this script if the others do more than you want.

Environment variables:

`JAVA`	The Java virtual machine to use. Default: The JVM detected during configuration.
`CLASSPATH`	Extra class path arguments for the JVM. Default: The Bees JAR file and the any of its dependencies.

6.2.2 bees

The `bees` script starts up a single blank node.

In addition to the environment variables read by `beesvm`, the bees script will read the following:

`BEESVM`	The command to use when starting up the JVM with the appropriate class path. Default: `beesvm`.
`BEESPORT`	The TCP port the Bees node should listen on for JSI messages. Default: 23000.
`VMPORT`	The TCP port the JanosVM should listen on for JSI messages. If you are not using the JanosVM, this will be ignored. Default: 24000.

6.2.3 nodesetup

The `nodesetup` script starts up one or more nodes, configures them using the `nodeconfig.py` script, and starts up an authenticator application. If more than one node is specified, the nodes will be started in separate xterm windows.

`BEESVM`	The command to use when starting up the JVM with the appropriate class path. Default: `beesvm`.
`JVMFLAGS`	Extra flags to pass to the JVM. Default: Empty.
`COUNT`	The number of nodes to start up. Default: 1
`IP0, IP1, IPN`	The IP address or hostname corresponding to a network interface to be used in discovery or normal communication. If multiple interfaces are to be used then each one must be specified as a different variable by incrementing the suffix number. Default: IP0 is set to the current host name.

`BROADCAST_IP`	The multicast IP address to use when multicasting discovery advertisements. Default: 239.192.0.50, an internal multicast address.
`BROADCAST_PORT`	The port to use when multicasting. Default: 8000
`PORT_BASE`	A base value for determining JSI port numbers. Bees ports start at the given value, while JanosVM port numbers start at this value plus 1000, and the node's neighbor port starts at plus 2000. If COUNT is greater than one, the value will be incremented by one after each node is started. Default: 23000
`NODE_START_INDEX`	If COUNT is greater than one, this value is used to uniquely name each node. If COUNT is equal to one, the node name will be the host name. Default: 0
`PYTHONPATH`	The python path to add to the standard Bees path. Default: The installed pylib directory.
`XTERM`	The xterm command line. Default: xterm -sb -sl 4000
`SYS_VERSION`	The version string bees.core.Node.getVersion() should return. Default: bees.

6.3 Introduction to the JanosVM Scripting Interface

This section presents a short description of the JanosVM Scripting Interface, or JSI. Originally developed for controlling the JanosVM, the interface worked well enough to be transferred over to Bees. The protocol is based primarily on the BeOS scripting interface which used system level messages to communicate data and instructions. Alternatively, JSI is designed for simplicity, any unreliable transport can be used, and the messages consist solely of strings. While this approach does poorly performance wise, the ease of use, flexibility, and reduced dependencies more than make up for the loss in speed. For instance, the simple `jsihey` program included with the JanosVM is able to control any JSI compatible program because it only needs to know how to build the messages from string given on the command line.

Bibliography

[1] Flux Research Group. An integrated experimental environment for distributed systems and networks. http://www.emulab.net/.

[2] James E. Housley. FreeBSD HEALTHD daemon software home page. http://healthd.thehousleys.net/.

[3] Yoshifumi R. Shimizu. FreeBSD XMBMON daemon software home page. http://www.nt.phys.kyushu-u.ac.jp/shimizu/download/download.html.