Annotated Source Code

The line numbers referenced throughout this document refer to the section entitled *The Scooby Source Code*, which starts on page 24.

It is recommended that this section be read *sequentially*, from start to finish. A number of techniques within the source code are repeated at various places within the code (and within the various parts of the system). When first encountered, each technique is described in detail. When subsequently encountered, the technique is referred to in a manner that assumes the reader is familiar with the detailed description[1].

1.1 The Key Server

The Key Server provides the facility with the ability to store and subsequently retrieve RSA Public Key values for the various entities within the environment: mobile agents and Locations.

The standard "magic" first line appears at 0001, and the programs version number is defined on line 0010.

Lines 0011-0018 import the Key Server's required modules, and lines 0019-0037 define a collection of constant values that are used throughout the source code.

A global variable[2], called `%allowed_connections` is defined as initially empty on line 0040. The `%allowed_connections` variable is a hash.

[1]The alternative would have been to repeat technical descriptions where appropriate, and unnecessarily expand the size of this section of the document. Too many trees would have died if this alternative had been followed.

[2]Within Perl, variables designated as "`our`" variables are lexically scoped to the file that contains them. This isn't really "global" as far as most other programming languages are concerned, but this is what "global" means in Perl.

A small *anonymous* subroutine is assigned to the `CHLD` signal handler on line 0042. This signal handler will be executed every time a child process signals its termination. This allows the parent process to successfully deal with "zombies"[3].

1.1.1 The Key Server's main code

The main code to the Key Server starts by processing the configuration file. Lines 0514-0523 open the configuration file (which is called `.keyserverrc` by default) and read it's contents into the `%allowed_connections` hash. Lines 0524-0528 displays status messages on screen (assuming the value of `ENABLED_PRINTS` is set to true, which it is by default). These messages indicate the list of hosts that the Key Server is willing to accept connections from.

Lines 0534-0544 establishes a connection with the MySQL database service which holds the `SCOOBY.publics` database table, aborting the Key Server if the connection cannot be established (on line 0543).

On lines 0548-0551 an SQL query is created to check the database table for an existing RSA Public Key for the Key Server. After forwarding the query to the database server on line 0552, lines 0553-0559 process the results returned from the database server, setting the `$pkplus_in_db` scalar to true if the RSA public key was found, false otherwise.

If not found, lines 0562-0591 generate an RSA Private/Public Key pairing (lines 0568-0575), storing the RSA Private/Public Keys in disk-files (line 0573). The RSA Public Key is read from its associated disk-file (lines 0580-0583), then used to build another SQL query to insert the the Key Server's RSA Public Key into the database table (lines 0584-0590).

If an RSA Public Key was found in the database table, a status message is displayed to this effect (lines 0594-0595) before closing the connection to the database server on line 0597.

The main code then forks a subprocess on line 0599. If this is unsuccessful, the Key Server aborts (line 0603). If successful, the child process starts the Web-based Monitoring Service on line 0608 (note the statement modifier on the call to the `_start_web_service`, which only starts the web-service if the `ENABLED_LOGGING` constant is defined).

The parent process (starting on line 0615) invokes the `fork` system call

[3]Zombie: a child process that has ended but has yet to have its process identifier removed from the operating systems process table.

again, creating *another* child process. Again, failure to do this results in the Key Server aborting on line 0619. Assuming success, the child process starts the Registration Service on line 0625, while the Responding Service is started by the parent process on line 0631.

`_start_web_service`, together with the `_start_registration_service` and `_start_responding_service` invocations are *never* returned from. Servers are permanent, in that they run forever (or until "killed" by the operating system).

Lines 0637-0709 are the Key Server's on-line documentation, which is printed in the appendix entitled *The On-line Documentation*.

1.1.2 _logger subroutine

Starting on line number: 0046.

This subroutine opens the logfile in append-mode (line 0057), then writes a single line to the file (line 0059). The line contains a time-stamp and the values of any parameters passed to this subroutine as arguments. The logfile is then closed (line 0060).

This subroutine is called by nearly every other subroutine in this program (including the main program code). It provides a convenient mechanism for logging the state of the Key Server.

1.1.3 _build_index_dot_html subroutine

Starting on line number: 0062.

This subroutine dynamically creates a file called "`index.html`" for use with the Web-based Monitoring Service. Any existing file is overwritten (line 0068), then a *HERE document* is used to write the new contents of the disk-file (line 0070-0099), before closing the disk-file on line 0100.

Note the inclusion of the name of the system running the Key Server within the generated HTML (line 0078), in addition to the current system time (line 0079). The content of the generated HTML page is taken from the Key Server's logfile, which is processed sequentially (lines 0085-0091).

1.1.4 _build_clearlog_dot_html subroutine

Starting on line number: 0102.

Similar to `_build_index_dot_html`, above, this code generates a HTML page for the Web-based Monitoring Service (called `clearlog.html`). A single argument is used to initialize a scalar (called `$backup_log`) on line 0108, before embedding the value in the HTML page on line 0118. Any existing HTML file of the same name is overwritten by this subroutine (line 0109).

1.1.5 _start_web_service subroutine

Starting on line number: 0128.

This subroutine implements a simple HTTP server[4] on a predefined protocol port number (line 0135). This code loops forever, waiting for a connection from a HTTP client[5] (line 0139). The HTTP server is based on the skeleton server provided by the `HTTP::Daemon` module.

When a connection arrives, the HTTP request is examined (line 0141). If the request method is "`GET`" (line 0144), the code checks to see if the request is for "`/`" or "`/index.html`" (line 0148) and, if it is, invokes the `_build_index_dot_html` subroutine to generated the requested HTML page before returning it to the HTTP client (lines 0151-0152).

If the request is for "`/clearlog.html`" (line 0154), the code backs up the existing logfile to an appropriately named file (lines 0157-0159), before removing the logfile from the file-system (line 0160).

The `_build_clearlog_dot_html` subroutine is then invoked to generate the requested HTTP resource, before returning it to the HTTP client (lines 0163 and 0163).

Any other HTTP request results in an appropriate error message returned to the HTTP client (lines 0167 and 0172).

1.1.6 _start_registration_service subroutine

Starting on line number: 0182.

The Registration Service begins by creating a socket-object to listen on (lines

[4] Commonly referred to as a "web server".
[5] Commonly referred to as a "web browser".

0199-0204), aborting on failure (line 0208). This subroutine then waits for connections from clients on line 0216.

When a connection arrives, line 0217 checks to see if the connection is originating from an allowed host. If it is NOT allowed, three things happen:

1. The unauthorized attempted connection is logged to the logfile (line 0219-0221).
2. A warning message prints on screen (lines 0222-0224), if appropriate (see the `ENABLED_PRINTS` constant).
3. A "go away" message is sent to the client (line 0225).

The connection to the client is then closed (line 0226), before starting another iteration (line 0227).

If the connection is allowed, a subprocess is created to service the client (line 0231). The subprocess (the child) registers an RSA Public Key with the database server on behalf of the client (lines 0233-0319). The IP address of the client (the peer) is determined on line 0238, then the protocol port number is received from the client connection (line 0240). The protocol port number is error-checked to ensure it conforms to an appropriate format (between 1 and 5 digits) on lines 0242 and 0243. If the format is NOT correct, the log is updated with the details (line 0246), a status message is printed (line 0247) and a "go away" message is sent to the client (line 0248). The connection to the client is then closed (line 0250), before starting another iteration (line 0253).

If the protocol port number is acceptable, a check is performed to ensure the client is not attempting to update the RSA Public Key of the Key Server (line 0255). If this client is attempting such an act of skulduggery, the usual fate (as describe above) awaits the client (lines 0257-0263).

Assuming an appropriate IP address and protocol port number, the code then receives (what it hopes is) an RSA Public Key from the client (line 0269). A connection to the database server is established (lines 0271-282), and an SQL SELECT query is created to determine if the database table already contains an entry for the IP address/protocol port number pairing (lines 0284-0287). The query is executed on line 0288, then the results are processed to determine whether the query returned results or not (lines 0293-0294). If an entry already exists, an SQL UPDATE query is created to perform the update (lines 0298-0303) or (if an entry does not exist) an SQL INSERT query is created to perform the insertion (lines 0308-0312). The query (which ever one is created) is executed on line 0314, then the client

exits on line 0318. This results in the generation of a `CHLD` signal, which is caught by the parent process and processed (removing any possibility of the child entering into a zombied state).

The Registration Service then iterates, ready for the next client connection.

1.1.7 _start_responder_service subroutine

Starting on line number: 0323.

The code in this subroutine is structured similarly to that of the previous subroutine, `_start_registration_service`. A number of techniques are common to both subroutines.

The Responding Service starts by creating a socket-object to communicate on (lines 0347-0352), aborting if this is NOT possible (line 0356).

An infinite loop is then entered into (line 0362), as servers are permanent. A connection is then waited for (line 0364). When one arrives, it is checked to see if it is allowed (line 0365), with appropriate action taken if the client is NOT authorized to connect (lines 0367-0373).

A subprocess is then created to service the client (line 0378), with the child process code on lines 0382 to 0502.

An IP address is received from the client connection (line 0385), followed by a protocol port number (line 0387). Lines 0389-0401 check to see if the IP address is appropriately formatted (line 0389) and takes the appropriate action if it is NOT formatted correctly (lines 0393-0400). Lines 0403-0415 check the format of the received protocol port number (line 0403), then takes the appropriate action if the protocol port number is NOT formatted correctly (lines 0407-0414).

Assuming a correctly formed IP address and protocol port number have been received, a connection is established with the MySQL database server (lines 0416-0421), aborting if the connection cannot be made (line 0425).

An SQL SELECT query is then formed (then executed) to extract the RSA Public Key associated with the received IP address and protocol port number from the `SCOOBY.publics` database table (lines 0430-0434). The response is then processed, starting on line 0439.

If the RSA Public Key for the Key Server was requested (line 0444), the string "`SELFSIG`", followed by the "`\n--end-sig--\n`" delimiter, defined by the `SIGNATURE_DELIMITER` constant (line 0453), followed by the RSA Public

Key for the Key Server (line 0456) is sent to the client.

If the request is for the RSA Public Key for some other entity, the RSA Public Key is signed (lines 0472-0476) by the Key Server's RSA Public Key, which was previously read-in from the Key Server's RSA Public Key disk-file (lines 0465-0469). The signature, followed by "`\n--end-sig--\n`" (line 0481), followed by the requested RSA Public Key (line 0484) is then sent to the client connection.

If the database server returned no results, the requested RSA Public Key was not found in the `SCOOBY.publics` database table. This fact is logged to the logfile (line 0489), then the string "`NOSIG`", followed by "`\n--end-sig--\n`" (line 0495), followed by the string "`NOTFOUND`" is sent to the client connection.

The client connection is then closed (line 0501) and the child process exits properly (line 0502), enabling the parent to check for, and subsequently, reap zombies.

The Responding Service then iterates, ready for the next client connection.

1.2 The `Executive.pm` Module

The `Executive.pm` module[6] is the smallest module within the facility. Despite its size, this source code plays a critical role within the environment.

This code is designed to be "used" by other Perl programs, so line 0710 declares the package name-space for this module to be "`Mobile::Executive`". Lines 0719-0732 are as required by the Perl module creation machinery. Note the automatic exportation of a single subroutine (on line 0723) and three scalar variables (lines 0724-0726). Five constant values are defined on line 0734-0738.

Perl is an interpreter. As such, source code is not executed until run-time (after compilation). This behaviour can be changed, in that it is possible to have some code execute at *compile-time*. This is accomplished by placing any compile-time code with a `BEGIN` block. Within modules, use of such a block allows some code to execute as soon as the module is "used" by another program. Within the `Executive.pm` source code, a `BEGIN` block (lines 0739-0756) does two things:

1. Determines the absolute path name of the program that is using the module (on line 0747).
2. Generates an RSA Public/Private Key Pairing for the program that is using this module (on lines 0748-0755).

The absolute path name of the program using `Executive.pm` is required by the Relocation Mechanism. The generated key-pairing is required by the Security Mechanism.

The on-line documentation to this module is located at the end of the source code file (lines 0776-0805).

1.2.1 relocate subroutine

Starting on line number: 0757.

This subroutine is never executed by the facility. The `Scooby.pm` module (described below) replaces the code in this subroutine with its own implementation of `relocate`. As such, the code in `Executive.pm`'s `relocate` subroutine acts simply as a placeholder for the "real" code that is executed

[6] More properly referred to as the `Mobile::Executive` module.

by the facility. Having said that, it is very important that programs that use this module provide the required parameters to the `relocate` call. Without these two values, the facility (nor anything else) can work out which Location and protocol port number to use for relocation.

1.3 The `Scooby.pm` Module

The `Scooby.pm` module implements the debugging facility that forms the "guts" of the facility. Unlike the other modules in the facility, the *Scooby* debugger is not able to take advantage of the `constant` module[7]. Consequently, the start of the source code defines a number of "`our`" variables as constants. These values are lexically scoped to the entire source code file[8] (lines 0829-0840). Although they are read/write by default, the intention is that they should be treated as read-only.

The on-line documentation to the module starts on line 1360 and extends through line 1408.

For a debugger to function within the Perl environment, it must be declared within the `DB` name-space. The `Scooby.pm` module does this on line 0845. All of the code within the `Scooby.pm` module "lives" within this name-space, which extends from line 0844 through 1355.

1.3.1 DB subroutine

Starting on line number: 0848.

This subroutine is called for every line in the program that can be breakpointed. Three scalar variables (that are "global" to the debugger) are set on line 0854: the current name-space (`$package`), the current filename (`$file`) and the current line number (`$line`). The latter two values are used by the `sub` subroutine (described below).

1.3.2 sub subroutine

Starting on line number: 0856.

This subroutine is called prior to every invocation of a subroutine call within the debugged program. It receives as parameters the same values are passed to the original subroutine call. The name of the current subroutine is assigned to the `$sub` scalar.

If the invoked subroutine call is to the `Mobile::Executive::relocate` subroutine (line 0864), code is executed to replace the invocation with replacement code (in lines 0866-1005). If the subroutine call is to any other named

[7] Just why this is so remains a complete mystery.
[8] Known as "global" in the Perl world.

subroutine, the `sub` subroutine first checks to see if the subroutine received any parameters (line 1009). If it did, the original subroutine is invoked with the parameters (line 1011) or without them (line 1015).

A collection of Perl modules are used by the debugger (lines 0866-0869). The first parameter to `relocate` is then assigned to the `$remote` scalar the used to determine the IP address (in dotted-decimal notation) of the next Location (lines 0871-0874). The second parameter to the original invocation of `relocate` is then assigned to the `$remote_port` scalar, then the name of the current file and - critically - the name of the *next line to execute* is remembered (lines 0875-0877).

Lines 0878-0890 determines the names of any "`my`" lexical variables currently being used within the calling scope (line 0881), freezes them (line 0883), before thawing them and converting the variables and their values into appropriately formatted Perl source code (lines 0886-0890). The `peek_my` subroutine is provided by the `PadWalker` module, and the `freeze` and `thaw` subroutines are provided by the `Storable` subroutine. The generated Perl source code is assigned to the `$stringified` scalar.

Lines 0892-0898 open the *Scooby* configuration file (line 0892) and extract the IP name (or address) of the Key Server (lines 0894-0897), ultimately assigning it to the `$key_server` scalar (line 0898).

With the name/address of the Key Server known, lines 0901-0902 contact the Key Server and request the RSA Public Key for the Key Server itself (0901), in addition to the next Location (0902).

Lines 0903-0923 perform the first mutation of the mobile agent's source code. The original source code disk-file is opened on line 0903. A temporary filename is created (line 0907), and then opened on line 0908. The original file is then read one-line-at-a-time, and written to the temporary disk-file (lines 0912-0921). When the current line count is equal to the remembered next line to execute (less one), the generated Perl source code (stored in the `$stringified` scalar) is inserted into the temporary disk-file (lines 0917-0920). Both disk-files are then closed (lines 0922-0923).

At this point, the original source code has been mutated to include the necessary Perl statements required to reinitialize any lexical variables after relocation.

The just-created temporary disk-file is then read back into memory (starting on line 0926), stored in the `@entire_toencrypt` array (0927), then deleted from local disk-storage (line 0931). Lines 0932-0950 encrypt the now memory-resident source code with the RSA Public Key of the next Location,

producing cyphertext (line 0939). This cyphertext is then digitally signed using the RSA Private Key of the mobile agent (lines 0946-0950). The next Location can use the RSA Public Key assigned to the mobile agent to verify the signature of any received cyphertext (which allows it to decide whether it is authentic or not), then decrypt the cyphertext with its own RSA Private Key.

Lines 0952-0966 begin the process of communicating with the next Location. A connection is established (line 0957), auto-flushing is switched on (lines 0960-0962), then the remembered filename and next line number are sent (lines 0964 and 0966, respectively).

Lines 0970-0995 register the RSA Public Key of the mobile agent with the Key Server. The key is first written to (lines 0976-0978), then read from the local disk-storage (lines 0979-0982). The is **deliberate**. The Key Server expects all RSA Public Key's to be in the written-to-disk-file-format produced by the `Crypt::RSA` module. Up until this point, the mobile agent's RSA Public Key has only existed in memory, so it needs to be written to a disk-file to force it into the expected format. Once read back in (lines 0979-0982), it is immediately deleted from the local disk-storage (line 0984) as it is no longer needed. A connection is established with the Key Server (lines 0986-0988), then the RSA Public Key is sent (lines 0993 and 0994), before closing the connection to the Key Server (line 0995).

The subroutine then waits for the Key Server to confirm that the key registration has been successful (line 0997).

After the confirmation, the digital signature and the cyphertext is sent to the next Location (on lines 0999 and 1001), before closing the connection (line 1002) and aborting the executing of the mobile agent (by calling `exit` on line 1004).

The mobile agent has been mutated, encrypted, digitally signed and sent to the next Location. It is no longer running on the machine that most recently executed it.

1.3.3 _wait_for_pkplus_confirm subroutine

Starting on line number: 1021.

This subroutine requests the mobile agent's RSA Public Key from the key Server. It returns when the Key Server successfully sends the key.

After using the `IO::Socket` module (line 1032), the three provided param-

eters are assigned to lexically scoped variables (lines 1034, 1035 and 1036). Starting from the assumption that the acknowledgment has not occurred yet (line 1038), the code loops until it is confirmed (lines 1039-1081.

A connection is established with the Key Server (lines 1042-1050), the request is sent (lines 1053-1054), a response is received (lines 1062-1064), then the connection is closed (line 1067). Line 1071 extracts the signature-part and the key-part from the message received from the Key Server, then a check on the signature-part is performed (line 1073). If the signature reads "`NOSIG`", the registration has not yet completed and another iteration is confirmed (line 1075). If the signature does not read "`NOSIG`", success is assumed, and the loop is allowed to end (line 1079).

The calling code can now continue, safe in the knowledge that the requested RSA Public Key exists within the Key Server.

1.3.4 _get_store_pkplus subroutine

Starting on line number: 1083.

This subroutine contacts the Key Server, requests a particular RSA Public Key, then stores the key within an appropriately named disk-file.

Two required modules (`Crypt::RSA` and `IO::Socket`) are used on line 1095 and 1096, respectively. The three required parameters are then assigned to lexically scoped variables (lines 1098-1100), then a connection to the Key Server is established (lines 1102-1140). The request is then sent to the Key Server (lines 1112-1113) and a response is received (lines 1118-1125).

Line 1129 extracts the signature-part and key-part from the message received from the Key Server, then the signature-part is checked. If it reads "`NOSIG`" (line 1130), the code aborts (line 1132) as no RSA Public Key has been found, which is a sure sign of trouble.

If the signature reads "`SELFSIG`" (line 1135), the RSA Public Key for the Key Server has been received, and it is stored to an appropriately named disk-file (lines 1136-1142).

Any other signature is verified against the RSA Public Key of the Key Server (lines 1146-1158). If the verification fails (line 1160), the program aborts (line 1162), otherwise the received RSA Public Key is stored in an appropriately named disk-file (lines 1166-1171).

1.3.5 _check_modules_on_remote subroutine

Starting on line number: 1176.

This subroutine contacts the next Location and tries to determine if a list of "used" classes exist on the next Location.

The three required parameters identify the next Location (lines 1186 and 1187) and provide the list of classes to check for (line 1188). Some standard Socket API code establishes a connection with the next Location (lines 1190-1201), then sends the list of classes to check (line 1204).

An alarm is set to expire in 10 seconds (lines 1212-1214), then the code attempts to receive a response from the next Location (lines 1217-1219). This code guards against the possibility of the next Location crashing during this phase of the relocation. If the alarm signals, an appropriate warning message is displayed (line 1231) upon expiration. If any other error occurred, the code aborts (line 1229). If a response is received before the alarm expires, the alarm is canceled (line 1219) and the response from the next Location is returned to the calling code (line 1234).

1.3.6 _storable_decode subroutine

Starting on line number: 1237.

This subroutine takes the "thawed" material from the `sub` subroutine and converts it into Perl source code.

In addition to the thawed material, this subroutine also receives as parameters the IP name/address and protocol port number that the next Location is running on (lines 1251-1253).

Three lexicals are then declared:

%for_refs - a hash containing the names and memory addresses of all referenced variables.

$stringified - a scalar which will end up containing the Perl source code that can be used to reinitialize the values of any lexicals within the current scope.

@required_classes - an array of class names for any objects existing within the current scope.

The thawed material is then iterated over twice.

The first pass extracts the values from the thawed material (which is stored within a hash called `%thawed`), assigning each variable name and value to scalars called (believe it or not) `$name` and `$value` (line 1268). Each iteration checks to see if the value associated with the variable name is referring to a scalar value (line 1270), an array (line 1283) or a hash (line 1288).

If the name refers to a scalar, the `%for_refs` hash is updated with the memory address of the scalar referred to (line 1272). Anything written to the `%for_refs` hash is used within the second pass (which is discussed below). A check to see if the scalar value is a number is performed (line 1274) in order to determine whether or not the scalar value needs to be enclosed within double quotes (line 1276) or not (line 1280). Note how the value associated with the scalar reference is extracted by accessing the value associated with the memory location referred to by `$value` (using `$$value`). Also, note how the `$stringified` scalar is "built up" with each iteration, but concatenating its previous contents with the additional Perl code within the `if` block[9].

If the name refers to an array, the `%for_refs` hash is updated with the memory address of the array referred to (line 1285). When an array is found, its value(s) are assigned to the name using the standard quote-words operator, `qw` (line 1286).

If the name refers to a hash, the `%for_refs` hash is updated with the memory address of the hash referred to (line 1290). The `$stringified` scalar is updated to include the code to reconstitute the hash on lines 1291-1296.

The second pass (beginning on line 1301) processes the `%thawed` hash again, this time in an effort to see if the value refers to an object (line 1304) or to a reference to another variable (line 1323).

If the value refers to an object, `Dumper` from the `Data::Dumper` module is used to convert the object into a textual form (line 1310). This textual form is then used to update the `$stringified` scalar (lines 1313-1319). Note how line 1306 pushes the name of the class onto the `@required_classes` array.

If the value refers to a reference, a reference to the variable name is added to the `$stringified` scalar (line 1325).

This subroutine ends by checking the next Location for the determined list of used classes (if there are any) on lines 1332-1350. If any of the required classes are missing, the code aborts on line 1343. If anything else occurs, the code aborts on line 1348. If all is well, the code continues and the value of

[9]The concatenation operator in Perl is: `.` (i.e., dot).

`$stringified` is returned to the calling code (line 1353).

The `$stringified` scalar now contains a series of Perl source code statements that can be used to reinitialize any lexical variables that were in scope immediately prior to the `relocate` invocation.

1.4 The `Location.pm` Class

The `Location.pm` class[10] provides for the creation of Locations within the environment that runs the facility. This Perl module is of the object-oriented variety - its a class.

This class is designed to be used by other Perl programs.

Lines 1426-1433 uses a collection of Perl modules, and line 1435 installs a signal handler for `CHLD` signals (ensuring that no child processes remain in a zombied state for any length of time). A series of constants are defined on lines 1437-1447, and two "global" variables are defined: `$VERSION` on line 1436, and `$_PWD` on line 1448. The former is required by the Perl module creation mechanisms, and the latter holds the initial working directory for the Location.

The on-line documentation to the module starts on line 2168 and extends through line 2246.

1.4.1 new method

Starting on line number: 1453.

This is the object's constructor, and it instantiates an `Mobile::Location` object (lines 1471-1472). The five settable attributes of the object are initialized (lines 1473-1477), then lines 1479-1481 untaint the Locations environment and path (for security reasons). Lines 1483-1485 determine, untaint and set the current working directory by assigning to the `$_PWD` "global" variable.

If the Location is running under superuser privilege (as 'root'), it aborts with an appropriate error message (line 1487). Additional attributes are set within the object (on lines 1491 and 1493). These values will be used later by the class methods.

The constructor concludes by spawning two subprocesses (on lines 1500 and 1502), before returning an object reference to the calling code (1503).

1.4.2 _logger method

Starting on line number: 1508.

[10] More properly referred to as the `Mobile::Location` class.

This method is very similar to the self-same named subroutine with the Key Server source code. A logfile is opened (line 1517), a message appended to it (line 1519) and then closed (line 1520).

1.4.3 _logger2 method

Starting on line number: 1522.

This method is practically identical to `_logger`, above. The only difference is that the logfile is opened in the directory immediately above the current one (note the double-dots on line 1533). The reason for this is that received mobile agents are executed in the "Location" directory which exists below the current working directory of the Location, and some status messages are written to the logfile immediately before re-execution begins.

1.4.4 _build_index_dot_html method

Starting on line number: 1538.

This method is very similar to that used within the Key Server source code, discussed above. An appropriately formed HTML page is created as a result of a request to the Web-based Monitoring Service. The content of the HTML page is drawn from the logfile.

1.4.5 _build_clearlog_dot_html method

Starting on line number: 1581.

This method is practically identical to the self-same named subroutine from the Key Server source code, displaying confirmation that the logfile has been cleared and a backup created.

1.4.6 _start_web_service method

Starting on line number: 1608.

This method is based on the `simplehttpd` web server from my book, which is itself based on the sample HTTP server included within the on-line documentation to the `HTTP::Daemon` module. This is also very similar to the

code within the Key Server source code, described above. Running on protocol port number 8080, a web server provides a mechanism to access the contents of the Location's logfile, as well as reset/backup it.

1.4.7 _register_with_keyserver method

Starting on line number: 1665.

This method generates an RSA Public/Private key-pairing (lines 1676-1686), stores a copy of the Location's RSA Private Key within the object's state (line 1689), then saves a copy of the RSA Public Key to an appropriately named disk-file (line 1693). After determining the IP name/address of the Key Server (lines 1695-1701), the Location then registers its RSA Public Key with the Key Server (lines 1705-1718).

1.4.8 start_concurrent method

Starting on line number: 1721.

This method creates a Location that process multiple relocations concurrently. A listening socket object is created (lines 1732-1739), then the Location is registered with the Key Server (line 1743). An infinite loop is started on line 1745, which waits for connections from clients. When one arrives (line 1747), a subprocess is created (line 1748) to service the relocation (line 1754). While the subprocess services the relocation, the parent process iterates and waits for the next client connection.

1.4.9 start_sequential method

Starting on line number: 1761.

This method creates a Location that process multiple relocations sequentially. A listening socket object is created (lines 1770-1777), then the Location is registered with the Key Server (line 1781). An infinite loop is started on line 1784, which waits for connections from clients. When one arrives (line 1786), this method services the relocation (line 1789), blocking any new connections. When done, the method iterates and waits for the next client connection.

1.4.10 _service_client method

Starting on line number: 1792.

This method is (by far) the longest subroutine within the entire facility, extending from line 1792 through to line 2042. It is the source code within this method that interacts, and communicates, with the `sub` subroutine from the `Scooby.pm` module (which implements the `relocate` invocation).

A single parameter is passed to this method: the socket object to communicate on (which is assigned to `$socket_object` on line 1800).

The mobile agent's filename is received from the connection (line 1801), then the filename-part (not the path) is extracted on line 1804. The next line number to execute is then received from the connection (line 1805). Lines 1807-1812 receive the digitally signed cyphertext, then line 1814 extracts the signature-part and the cyphertext-part of the received message.

Lines 1817-1840 contact the Key Server and request the RSA Public Key of the communicating mobile agent. The received (digitally signed) key is broken into its signature-part and key-part (line 1844). If the received signature reads "`NOSIG`" (line 1845), the Location aborts (line 1852) after closing the connection to the client (line 1851).

Assuming a verified digital signature, the mobile agent's RSA Public Key is written to an appropriately named disk-file (lines 1854-1857). This key is then used to verify the digital signature of the received cyphertext (lines 1864-1867), aborting if the mobile agent's digital signature is not verified (line 1873).

If verification succeeds, the Location uses its own RSA Private Key to decrypt the received cyphertext (lines 1881-1885), aborting if the decryption was not successful (line 1889). Assuming success, the decrypted source code (referred to as `$plaintext` within this method) is converted from a "flat" string into an array of new-line terminated lines (lines 1893-1898). Converting to this format greatly simplifies the mutation that the Location is required to perform on the received source code. The source code now exists in an array called `@entire_thing`.

Lines 1900-1910 ensure that the Location is working within the correct subdirectory, creating a new directory (within which to work) if needs be.

If the creator of the Location has switched on agent logging, lines 1912-1922 save a copy of the pre-mutated source code to a disk-file with an appropriately unique name (line 1914).

A disk-file with the same base name as the received mobile agent is then created within the disk-storage of the Location (line 1928). A unique label is generated (line 1931), then the `@entire_thing` array is processed one-line-at-a-time (starting on line 1934).

The "magic" first line is written to the disk-file (line 1936).

Lines 1937-2003 have been "commented-out" of this version of this method. If activated, this code would add instructions to the disk-file that would arrange to execute the source code within a restricted compartment. As discussed in the *Design Deviations* section of this document, this functionality cannot be enabled at this time. However, the source code is ready-and-waiting to be activated once a fix emerges from the `Crypt::RSA` developers.

The just-generated label is written to the disk-file (line 2006) as the parameter to the dreaded `goto` statement. A variable called `$line_counter` is then initialized to the value 2. This variable is set to this value to keep the line count synchronized with the next-line-number-to-execute value received from the mobile agent. The rest of the lines of source code within the `@entire_thing` array are then processed sequentially (lines 2008-2019), before closing the disk-file on line 2021. Note that the label is added to the disk-file (line 2014) within the loop whenever the code determines that it has just written the line that contains the `relocate` invocation. This addition, as well as the `goto` statement at the start of the disk-file, supports the re-execution of the mobile agent from where it left off when next executed.

A command-line is formed (line 2030), the connection with the mobile agent is closed (line 2033) and the mobile agent is re-executed using the just formed command-line (line 2039). Any results returned from the invocation of the command-line are displayed on the Location's display (line 2041).

The mobile agent has been received, its digital signature has been verified, its source code decrypted, mutated and then executed.

1.4.11 _spawn_web_monitoring_service method

Starting on line number: 2043.

This method simple spawns a subprocess (line 2050) and starts the Web-based Monitoring Service within the spawned process (line 2055), assuming the value associated with the `Web` attribute within the object is set to true.

1.4.12 _generate_label support subroutine

Starting on line number: 2062.

This subroutine is not part of the object. It is not designed to be invoked through the object, as it solely exists to provide a support service to the object methods described above.

This subroutine takes three values (lines 2070-2072), sanitizes one of them to remove any unwanted characters (line 2074), the combines the three values with the word "LABEL_" to produce a (hopefully) unique label.

1.4.13 _check_for_modules support subroutine

Starting on line number: 2077.

This subroutine is not part of the object. It is not designed to be invoked through the object, as it solely exists to provide a support service to the object methods described above.

Given a list of Perl modules to check for (line 2086), this subroutine cycles through them (lines 2088-2098) and checks to see if they are installed within the Location's Perl system (line 2091) and, if it does not, adds the name of the module to an array (line 2097).

This subroutine ends by returning the list of modules not found on line 2100.

1.4.14 _spawn_network_service support subroutine

Starting on line number: 2102.

This subroutine is not part of the object. It is not designed to be invoked through the object, as it solely exists to provide a support service to the object methods described above.

This subroutine creates a subprocess that waits to be contacted at a predetermined protocol port number, which is passed as a parameter (line 2109). Once contacted, the code treats the message received as a list of modules. This list of modules are then checked to see if they exist within the Location's Perl system.

Line 2114 creates a subprocess, then the `if` block (lines 2117-2157) implements a network server (within the subprocess) that waits for connections

from a mobile agent (line 2132), receives a message (lines 2138-2144), turns the message into a list of module names (line 2145), then checks for their existence (line 2146). If any of the modules are NOT found (line 2147), the string "`NOK: `" together with the list of not found modules are returned the mobile agent over the network connection (line 2149). If all the modules exist, the string "`OK`" is returned to the mobile agent (line 2153). The network connection to the mobile agent is then closed (line 2158).

1.5 The Scooby Source Code

This appendix presents the entire source code to the facility. Each non-blank line is numbered. For type-setting purposes, some lines are extended over more than one line in order to fit within the printed page. Such lines still only warrant an individual line number.

1.6 The Key Server Source Code

```
0001  #! /usr/bin/perl -w

0002  # keyserver - The Responder/Registration Public-Key Service for use with the
0003  #             Devel::Scooby, Mobile::Executive and Mobile::Location modules.
0004  #
0005  # Author:  Paul Barry, paul.barry@itcarlow.ie
0006  # Create:  April 2003.
0007  # Update:  May 2003 - added support for new protocol_port field in database.
0008  #                   - added support for logging to the LOGFILE.
0009  #                   - added support for HTTP web-based monitoring.

0010  our $VERSION = 1.04;

0011  use strict;

0012  use Crypt::RSA;       # Provides signing service for authentication.
0013  use HTTP::Daemon;     # Provides a basic HTTP server.
0014  use HTTP::Status;     # Provides support for HTTP status messages.
0015  use IO::Socket;       # Provides OO interface to TCP/IP sockets API.
0016  use Net::MySQL;       # Allows for direct communications with MySQL db.
0017  use POSIX 'WNOHANG';  # Ensures POSIX-compliant handling of "zombies".
0018  use Sys::Hostname;    # Provides a means of determining the name of machine.

0019  use constant KEYSRV_PASSWD       => 'keyserver';
0020  use constant KEY_SIZE            => 1024;

0021  use constant ENABLED_LOGGING     => 1;  # Set to 0 to disable logging to
                                              LOGFILE.
0022  use constant ENABLED_PRINTS      => 1;  # Set to 0 to disable screen
                                              messages.

0023  use constant SIGNATURE_DELIMITER => "\n--end-sig--\n";

0024  use constant HTML_DEFAULT_PAGE   => "index.html";
0025  use constant HTTP_PORT           => 8080;

0026  use constant CONFIGHOSTS_FILE    => '.keyserverrc';

0027  use constant RESPONDER_PPORT     => '30001';
0028  use constant REGISTRATION_PPORT  => '30002';

0029  use constant LOCALHOST           => '127.0.0.1';

0030  use constant KEYDB_HOST          => 'localhost';
0031  use constant KEYDB_DB            => 'SCOOBY';
0032  use constant KEYDB_USER          => 'perlagent';
0033  use constant KEYDB_PASS          => 'passwordhere';

0034  use constant TRUE                => 1;
0035  use constant FALSE               => 0;

0036  use constant LOGFILE             => 'keyserver.log';

0037  use constant VISIT_SCOOBY        => 'Visit the <a href="http://glasnost.
                                          itcarlow.ie/~scooby/">Scooby Website
                                          </a> at IT Carlow.<p>';

0038  # The "%allowed_connections" hash is written to during the start-up phase
0039  # of this program.  It is referred to later, but should NEVER be written to.
```

```
0040  our %allowed_connections = (); # XXXXX: this is a 'global'.

0041  # Install a signal-handler to kill off "zombies" should they arise.

0042  $SIG{CHLD} = sub { while ( ( my $kid = waitpid( -1, WNOHANG ) ) > 0 ) { } };

0043  ##############################################################################
0044  # Support subroutines start here.
0045  ##############################################################################

0046  sub _logger {

0047      # This small routine quickly writes a message to the LOGFILE.  Note that
0048      # every line written to the LOGFILE is timestamped.
0049      #
0050      # Note: a more "efficient" implementation would open the LOGFILE when
0051      # the keyserver starts up then append to it as required.  This method
0052      # will do for now.

0053      # IN:   a message to log.
0054      #
0055      # OUT:  nothing.

0056      # Open the LOGFILE for append >>.

0057      open KEY_LOGFILE, ">>" . LOGFILE
0058          or die "keyserver: unable to append to this keyserver's LOGFILE.\n";

0059      print KEY_LOGFILE scalar localtime, ": @_\n";

0060      close KEY_LOGFILE;
0061  }

0062  sub _build_index_dot_html {

0063      # Builds the INDEX.HTML file (used by _start_web_service).
0064      #
0065      # IN:  nothing.
0066      #
0067      # OUT: nothing (although "index.html" is created).

0068      open HTMLFILE, ">index.html"
0069          or die "Fatal error: index.html cannot be written to: $!.\n";

0070      print HTMLFILE<<end_html;

0071  <HTML>
0072  <HEAD>
0073  <TITLE>Welcome to the Key Server's Web-Based Monitoring Service.</TITLE>
0074  </HEAD>
0075  <BODY>
0076  <h2>Welcome to the Key Server's Web-Based Monitoring Service</h2>
0077  end_html

0078      print HTMLFILE "Key Server running on: <b>" . hostname() . "</b>.<p>";
0079      print HTMLFILE "Key Server date/time: <b>" . localtime() . "</b>.<p>";
0080      print HTMLFILE<<end_html;

0081  Click <a href="clearlog.html">here</a> to reset the log.
0082  <h2>Logging Details</h2>
0083  <pre>
```

```
0084  end_html

0085      open HTTP_LOGFILE, LOGFILE
0086          or die "keyserver: the LOGFILE is missing - aborting.\n";

0087      while ( my $logline = <HTTP_LOGFILE> )
0088      {
0089          print HTMLFILE "$logline";
0090      }

0091      close HTTP_LOGFILE;

0092      print HTMLFILE<<end_html;

0093  </pre>
0094  end_html

0095      print HTMLFILE VISIT_SCOOBY;
0096      print HTMLFILE<<end_html;

0097  </BODY>
0098  </HTML>
0099  end_html

0100      close HTMLFILE;
0101  }

0102  sub _build_clearlog_dot_html {

0103      # Builds the CLEARLOG.HTML file (used by _start_web_service).
0104      #
0105      # IN:  the name of the just-created backup file.
0106      #
0107      # OUT: nothing (although "clearlog.html" is created).

0108      my $backup_log = shift;

0109      open CLEARLOG_HTML, ">clearlog.html"
0110          or die "Fatal error: clearlog.html cannot be written to: $!.\n";

0111      print CLEARLOG_HTML<<end_html;

0112  <HTML>
0113  <HEAD>
0114  <TITLE>Key Server's Logfile Reset.</TITLE>
0115  </HEAD>
0116  <BODY>
0117  <h2>Key Server's Logfile Reset</h2>
0118  The previous logfile has been archived as: <b>$backup_log</b><p>
0119  Return to the Key Server's <a href="index.html">main page</a>.<p>
0120  end_html

0121      print CLEARLOG_HTML VISIT_SCOOBY;
0122      print CLEARLOG_HTML<<end_html;

0123  </BODY>
0124  <HTML>
0125  end_html

0126      close CLEARLOG_HTML;
0127  }
```

```
0128  sub _start_web_service {

0129      # Starts a small web server running on port HTTP_PORT.  Provides for
0130      # some simple monitoring of the keyserver.
0131      #
0132      # IN:  nothing.
0133      #
0134      # OUT: nothing.

0135      my $httpd = HTTP::Daemon->new( LocalPort => HTTP_PORT,
0136                                     Reuse     => 1 )
0137          or die "keyserver: could not create HTTP daemon on " .
0138                      HTTP_PORT . ".\n";

0139      while ( my $http_client = $httpd->accept )
0140      {
0141          if ( my $service = $http_client->get_request )
0142          {
0143              my $request = $service->uri->path;

0144              if ( $service->method eq 'GET' )
0145              {
0146                  my $resource;
0147
0148                  if ( $request eq "/"  || $request eq "/index.html" )
0149                  {
0150                      $resource = HTML_DEFAULT_PAGE;

0151                      _build_index_dot_html;

0152                      $http_client->send_file_response( $resource );
0153                  }
0154                  elsif ( $request eq "/clearlog.html" )
0155                  {
0156                      # Create a name for the backup log.

0157                      my $backup_log = "keyserver." . localtime() . ".log" ;

0158                      # Make the backup, delete the LOGFILE, then recreate it.

0159                      system( "cp", LOGFILE, $backup_log ) ;
0160                      unlink LOGFILE;
0161                      _logger( "KEYSERVER: log reset." ) if ENABLED_LOGGING;

0162                      _build_clearlog_dot_html( $backup_log );

0163                      $http_client->send_file_response( "clearlog.html" );
0164                  }
0165                  else
0166                  {
0167                      $http_client->send_error( RC_NOT_FOUND );
0168                  }
0169              }
0170              else
0171              {
0172                  $http_client->send_error( RC_METHOD_NOT_ALLOWED );
0173              }
0174          }
0175      }
0176      continue
0177      {
0178          $http_client->close;
```

```
0179          undef( $http_client );
0180      }
0181  }

0182  sub _start_registration_service {

0183      # The Registration Service waits passively at protocol port number
0184      # REGISTRATION_PPORT for TCP-based connections.  When one arrives,
0185      # the IP address of the client is determined, a protocol port number is
0186      # received, followed by a PK+.  These values are either added to the
0187      # 'SCOOBY.publics' table or used to update an existing entry in
0188      # the 'SCOOBY.publics' table.
0189      #
0190      # A request to add LOCALHOST and RESPONDER_PPORT to the database is
0191      # REJECTED, as these values are used by the keyserver to store it's own
0192      # PK+.

0193      # No ACK is provided to the client.  Clients can use the Responder
0194      # Service to check that their PK+ has been added to the database.
0195      #
0196      # IN:  nothing.
0197      #
0198      # OUT: nothing.

0199      my $registration_socket = IO::Socket::INET->new(
0200                                    LocalPort => REGISTRATION_PPORT,
0201                                    Listen    => SOMAXCONN,
0202                                    Proto     => 'tcp',
0203                                    Reuse     => TRUE
0204                                );

0205      if ( !defined( $registration_socket ) )
0206      {
0207          _logger( "REGISTRATION: could not create initial socket - fatal." )
                  if ENABLED_LOGGING;

0208          die "keyserver: (registration): could not create socket: $!.\n";
0209      }

0210      print "The Registration Service is starting up on port: ",
0211                $registration_socket->sockport, "\n" if ENABLED_PRINTS;

0212      _logger( "REGISTRATION: up on port: " . $registration_socket->sockport .
              "." ) if ENABLED_LOGGING;

0213      # Servers are permanent - they NEVER end.

0214      while ( TRUE )
0215      {
0216          next unless my $from_socket = $registration_socket->accept;

0217          if ( !exists
                     $allowed_connections{ inet_ntoa( $from_socket->peeraddr ) } )
0218          {
0219              _logger( "REGISTRATION: unauthorized host " .
0220                  inet_ntoa( $from_socket->peeraddr ) .
0221                      " rejected." ) if ENABLED_LOGGING;

0222              print "Warning: request from an unauthorized host (" .
0223                        inet_ntoa( $from_socket->peeraddr ) .
0224                            ") rejected.\n" if ENABLED_PRINTS;
```

```
0225            print $from_socket "keyserver: you are NOT permitted to talk:
                      disconnecting ... \n";

0226            $from_socket->close;

0227            next;
0228        }

0229        # Create a sub-process to serve client.

0230        _logger( "REGISTRATION: creating subprocess." ) if ENABLED_LOGGING;

0231        next if my $pid = fork;
0232
0233        if ( $pid == 0 )
0234        {
0235            # The registration socket is not needed in child so it's closed.

0236            $registration_socket->close;

0237            # Determine the IP address of the other end of the socket.

0238            my $peer_ip = inet_ntoa( $from_socket->peeraddr );

0239            # Receive the protocol port number from the socket.

0240            my $peer_port = <$from_socket>;

0241            # Untaint the value of "$peer_port", using a regex.

0242            $peer_port =~ /^(\d{1,5})$/;
0243            $peer_port = $1;

0244            if ( !defined( $peer_port ) )
0245            {
0246                _logger( "REGISTRATION: invalid protocol port received from
                          $peer_ip." ) if ENABLED_LOGGING;

0247                print "Warning: invalid protocol port received - request
                          ignored.\n" if ENABLED_PRINTS;

0248                print $from_socket "keyserver: you sent an invalid protocol
                          port number - disconnecting ... \n";

0249                # No more client interaction.

0250                close $from_socket;

0251                # Short-circuit as it is not possible to continue without a
0252                # valid protocol port number.

0253                next;
0254            }

0255            if ( $peer_ip eq LOCALHOST && $peer_port eq RESPONDER_PPORT )
0256            {
0257                _logger( "KEYSERVER: attempt to add PK+ for keyserver to
                          database - ignored." ) if ENABLED_LOGGING;

0258                print "Warning: attempt to add PK+ for keyserver to database
                          - ignored.\n" if ENABLED_PRINTS;
```

```
0259                print $from_socket "You cannot update the keyserver's PK+
                          - disconnecting ... \n";

0260                # No more client interaction.

0261                close $from_socket;

0262                # Short-circuit: LOCALHOST and RESPONDER_PPORT are RESERVED.

0263                next;
0264            }

0265            # Note: we blindly trust that the client does indeed send a
0266            # PK+ value.  It's perhaps more prudent to check the PK+
0267            # before adding it to the database?  Ah, time, if only I
0268            # had more of it ...

0269            my @entire_key = <$from_socket>;

0270            close $from_socket;

0271            my $connection = Net::MySQL->new(
0272                                 hostname  => KEYDB_HOST,
0273                                 database  => KEYDB_DB,
0274                                 user      => KEYDB_USER,
0275                                 password  => KEYDB_PASS
0276                             );

0277            if ( $connection->is_error )
0278            {
0279               _logger( "REGISTRATION: could not contact database - fatal." )
                         if ENABLED_LOGGING;

0280                die "keyserver: (registration): " .
0281                    $connection->get_error_message . ".\n";
0282            }

0283            # Check to see if we need to do an INSERT or an UPDATE.

0284            my $query = 'select ip_address ' .
0285                        'from publics where ' .
0286                        "( ip_address = \"$peer_ip\" and " .
0287                        "protocol_port = \"$peer_port\" )";

0288            $connection->query( $query );

0289            _logger( "REGISTRATION: querying DB for existing
                      $peer_ip/$peer_port combination." ) if ENABLED_LOGGING;

0290            # This next line suppresses the warning messages from
0291            # the Net::MySQL module - they are NOT needed/wanted here.

0292            local $SIG{__WARN__} = sub {}; # Comment-out this line when
                      testing.

0293            my $iterator = $connection->create_record_iterator;
0294            my $rec = $iterator->each;

0295            if ( ref( $rec ) eq 'ARRAY' )
0296            {
0297                # The ip_address/protocol-port/key already exist, so do
                        an UPDATE.
```

```
0298                 _logger ( "REGISTRATION: updating $peer_ip/$peer_port." )
                           if ENABLED_LOGGING;

0299                 print "[UPDATE] Updating the PK+ for $peer_ip/$peer_port.\n"
                           if ENABLED_PRINTS;

0300                 $query = 'update publics set ' .
0301                          "public_key = \"@entire_key\" where " .
0302                          "( ip_address = \"$peer_ip\" and " .
0303                          "protocol_port = \"$peer_port\" )";
0304             }
0305             else
0306             {
0307                 # The ip_address/protocol-port/key are new, so do an INSERT.

0308                 _logger ( "REGISTRATION: inserting $peer_ip/$peer_port." )
                           if ENABLED_LOGGING;

0309                 print "[INSERT] Inserting the $peer_ip/$peer_port pairing.\n"
                           if ENABLED_PRINTS;

0310                 $query = 'insert into publics ' .
0311                     '( ip_address, protocol_port, public_key ) values ' .
0312                     "( \"$peer_ip\", \"$peer_port\", \"@entire_key\" )";
0313             }

0314             $connection->query( $query );

0315             # We assume a successful insert/update, which may be a little
0316             # naive. Of course, the client can always use the Responder
0317             # Service to check the state of the database, if required.

0318             exit 0;
0319         }

0320         $from_socket->close;
0321     }
0322 }

0323 sub _start_responder_service {

0324     # The Responder Service waits passively at protocol port number
0325     # RESPONDER_PPORT for TCP-based connections. When one arrives,
0326     # the IP address of the client is determined, then an IP address and
0327     # protocol port number is received. These are then used to look-up a PK+
0328     # from the 'SCOOBY.publics' table. If a PK+ is found in the database,
0329     # it is read from the 'SCOOBY.publics' table, signed by the
0330     # keyserver, then sent to the client. If the PK+ is NOT found, the
0331     # string 'NOSIG' followed by 'NOTFOUND' is sent to the client.
0332     #
0333     # Note: the PK+ is signed, but NOT encrypted. There is no need to
0334     # add a further level of security.  The signature is enough, and the
0335     # PK+ is a public key, after all.
0336     #
0337     # If a request is received for IP address LOCALHOST and protocol port
0338     # RESPONDER_PPORT, then the PK+ is looked-up and sent UNSIGNED.  This is
0339     # due to the fact that it does not make sense to sign the PK+ for
0340     # the keyserver, as the client most likely needs the PK+ to verify
0341     # signatures.  The string "SELFSIG" (followed by the PK+) is sent in
0342     # this case.
0343     #
```

```
0344      # IN:  nothing.
0345      #
0346      # OUT: nothing.

0347      my $responder_socket = IO::Socket::INET->new(
0348                                  LocalPort => RESPONDER_PPORT,
0349                                  Listen    => SOMAXCONN,
0350                                  Proto     => 'tcp',
0351                                  Reuse     => TRUE
0352                              );

0353      if ( !defined( $responder_socket ) )
0354      {
0355          _logger( "RESPONDER: could not create initial socket - fatal." )
                  if ENABLED_LOGGING;

0356          die "keyserver: (responder): could not create socket: $!.\n";
0357      }

0358      print "The Responder Service is starting up on port: ",
0359                $responder_socket->sockport, "\n" if ENABLED_PRINTS;

0360      _logger( "RESPONDER: up on port: " . $responder_socket->sockport . "." )i
              if ENABLED_LOGGING;

0361      # Servers are permanent - they NEVER end.

0362      while ( TRUE )
0363      {
0364          next unless my $from_socket = $responder_socket->accept;

0365          if ( !exists
                       $allowed_connections{ inet_ntoa( $from_socket->peeraddr ) } )
0366          {
0367              _logger( "RESPONDER: unauthorized host " .
0368                        inet_ntoa( $from_socket->peeraddr ) .
0369                            " request rejected." ) if ENABLED_LOGGING;

0370              print "Warning: request from an unauthorized host (" .
0371                        inet_ntoa( $from_socket->peeraddr ) .
0372                            ") rejected.\n" if ENABLED_PRINTS;

0373              print $from_socket "keyserver: you are NOT permitted to talk:
                        disconnecting ... \n";

0374              $from_socket->close;

0375              next;
0376          }

0377          # Create a sub-process to serve client.

0378          next if my $pid = fork;
0379
0380          if ( $pid == 0 )
0381          {
0382              # The Responder Socket is not needed in child, so it's closed.

0383              $responder_socket->close;

0384              # Receive the IP address and protocol port number to lookup.
```

```
0385            my $ip_lookup = <$from_socket>;

0386            chomp( $ip_lookup );

0387            my $port_lookup = <$from_socket>;

0388            # Untaint the value of "$ip_lookup", using a regex.

0389            $ip_lookup =~ /^(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})$/;
0390            $ip_lookup = $1;

0391            if ( !defined( $ip_lookup ) )
0392            {
0393                _logger ( "RESPONDER: invalid IP address sent - request
                        ignored." ) if ENABLED_LOGGING;

0394                print "Warning: invalid IP address sent to Responder Service
                        - request ignored.\n" if ENABLED_PRINTS;

0395                print $from_socket "keyserver: you sent an invalid IP
                        address - disconnecting ... \n";

0396                # No more client interaction.

0397                close $from_socket;

0398                # Short-circuit as it is not possible to continue without a
0399                # valid IP address.

0400                next;
0401            }

0402            # Untaint the value of "$port_lookup", using a regex.

0403            $port_lookup =~ /^(\d{1,5})$/;
0404            $port_lookup = $1;

0405            if ( !defined( $port_lookup ) )
0406            {
0407                _logger( "RESPONDER: invalid protocol port sent by
                        $ip_lookup - request ignored." ) if ENABLED_LOGGING;

0408                print "Warning: invalid protocol port sent to Responder
                        Service - request ignored.\n" if ENABLED_PRINTS;

0409                print $from_socket "keyserver: you sent an invalid protocol
                        port number - disconnecting ... \n";

0410                # No more client interaction.

0411                close $from_socket;

0412                # Short-circuit as it is not possible to continue without a
0413                # valid protocol port number.

0414                next;
0415            }

0416            my $connection = Net::MySQL->new(
0417                                hostname  => KEYDB_HOST,
0418                                database  => KEYDB_DB,
0419                                user      => KEYDB_USER,
```

```
0420                                password  => KEYDB_PASS
0421                            );

0422            if ( $connection->is_error )
0423            {
0424               _logger( "RESPONDER: could not contact database - fatal." )
                         if ENABLED_LOGGING;

0425                die "keyserver: (responder): " .
0426                    $connection->get_error_message . ".\n";
0427            }

0428            print "Checking the PK+ value for $ip_lookup/$port_lookup.\n"
                      if ENABLED_PRINTS;

0429            # Check to see if the ip_address/protocol port exists in the db.

0430            my $query = 'select public_key ' .
0431                        'from publics where ' .
0432                        "( ip_address = \"$ip_lookup\" and " .
0433                        "protocol_port = \"$port_lookup\" )";

0434            $connection->query( $query );

0435            _logger( "RESPONDER: querying DB for existing
                      $ip_lookup/$port_lookup combination." ) if ENABLED_LOGGING;

0436            # This next line suppresses the warning messages from
0437            # the Net::MySQL module - they are NOT needed/wanted here.

0438            local $SIG{__WARN__} = sub {}; # Comment-out this line when
                      testing.

0439            my $iterator = $connection->create_record_iterator;
0440            my $rec = $iterator->each;

0441            # If the ip_address/protocol-port/key exist, send the PK+.
0442            if ( ref( $rec ) eq 'ARRAY' )
0443            {
0444                if ( $ip_lookup eq LOCALHOST && $port_lookup eq
                             RESPONDER_PPORT )
0445                {
0446                    _logger( "RESPONDER: sending my PK+ to " .
0447                        inet_ntoa( $from_socket->peeraddr ) .  "." )
                                 if ENABLED_LOGGING;

0448                    print "  -> No need to sign PK+ for keyserver.\n"
                             if ENABLED_PRINTS;
0449                    print "  --> Sending SELFSIG to client (" .
0450                              inet_ntoa( $from_socket->peeraddr ) .
0451                                  ").\n" if ENABLED_PRINTS;

0452                    # The ip_address is that of the keyserver, so send
                             "SELFSIG".
0453                    print $from_socket "SELFSIG" . SIGNATURE_DELIMITER;

0454                    print "  ---> Sending PK+ for $ip_lookup/$port_lookup
                             to client.\n" if ENABLED_PRINTS;

0455                    # Send the keyserver's PK+ to the client.

0456                    print $from_socket "$rec->[0]";
```

```
0457                }
0458                else
0459                {
0460                    _logger( "RESPONDER: sending PK+ for
                                $ip_lookup/$port_lookup to " .
0461                                  inet_ntoa( $from_socket->peeraddr ) .  "." )
                                        if ENABLED_LOGGING;

0462                    print "  -> Signing PK+ for $ip_lookup/$port_lookup.\n"
                                if ENABLED_PRINTS;

0463                    # Get the PK- from it's disk-file.
0464                    my $ksf = LOCALHOST . '.' .RESPONDER_PPORT. '.private';

0465                    my $pkminus = new Crypt::RSA::Key::Private(
0466                                            Filename => $ksf,
0467                                            Password => KEYSRV_PASSWD,
0468                                            Armour   => TRUE
0469                                          );

0470                    my $rsa = new Crypt::RSA;

0471                    # Use the PK- to sign the PK+.

0472                    my $signature = $rsa->sign(
0473                                            Message => $rec->[0],
0474                                            Key     => $pkminus,
0475                                            Armour  => TRUE
0476                                          );

0477                    print "  --> Sending signature to client (" .
0478                              inet_ntoa( $from_socket->peeraddr ) .
0479                                  ").\n" if ENABLED_PRINTS;

0480                    # Send the printable signature to the client.

0481                    print $from_socket "$signature" . SIGNATURE_DELIMITER;

0482                    print "  ---> Sending PK+ for $ip_lookup/$port_lookup
                                to client.\n" if ENABLED_PRINTS;

0483                    # Send the PK+ to the client.

0484                    print $from_socket "$rec->[0]";
0485                }
0486            }
0487            else
0488            {
0489                _logger( "RESPONDER: sending NOSIG/NOTFOUND for
                              $ip_lookup/$port_lookup to " .
0490                                inet_ntoa( $from_socket->peeraddr ) .  "." )
                                      if ENABLED_LOGGING;

0491                print "  -> Sending NOSIG to client (" .
0492                                  inet_ntoa( $from_socket->peeraddr ) .
0493                                      ").\n" if ENABLED_PRINTS;

0494                # The ip_address/protocol-port does not exist, send "NOSIG".

0495                print $from_socket "NOSIG" . SIGNATURE_DELIMITER;
0496
0497                print "  --> Sending NOTFOUND for $ip_lookup/$port_lookup
```

```
                        to client.\n" if ENABLED_PRINTS;

0498                # The ip_address/protocol-port does not exist, send
                        "NOTFOUND".

0499                print $from_socket "NOTFOUND";
0500            }

0501            $from_socket->close;

0502            exit 0;
0503        }

0504        # Not needed in the parent's code, so it is closed.

0505        $from_socket->close;
0506    }
0507 }

0508 ###############################################################################
0509 # Main code starts here
0510 ###############################################################################

0511 # Start by populating the "%allowed_connections" hash from the keyserver's
0512 # configuration file.  Connections from every other IP address/port are
0513 # ignored/rejected.

0514 open CONFIGFILE, CONFIGHOSTS_FILE
0515     or die "keyserver: the .keyserverrc configuration file does not exist:
             $!.\n";

0516 while ( my $line = <CONFIGFILE> )
0517 {
0518     chomp( $line );

0519     my ( $host, $port ) = split /:/, $line;
0520
0521     $allowed_connections{ $host } = $port;
0522 }

0523 close CONFIGFILE;

0524 print "Accepting connections/requests from:\n" if ENABLED_PRINTS;

0525 while ( my ( $host, $port ) = each %allowed_connections )
0526 {
0527     print "  -> $host on port(s): $port.\n" if ENABLED_PRINTS;
0528 }

0529 # Prior to starting the network servers, check the database to see if a
0530 # PK+ value exists for itself (using address LOCALHOST).  If it does,
0531 # then things are fine-and-dandy.  If the PK+ is missing, both the
0532 # PK- and PK+ keys are regenerated and the database/disk-files updated.

0533 # Begin by opening a new connection to the database.

0534 my $connection = Net::MySQL->new(
0535                     hostname  => KEYDB_HOST,
0536                     database  => KEYDB_DB,
0537                     user      => KEYDB_USER,
0538                     password  => KEYDB_PASS
0539                 );
```

```
0540  if ( $connection->is_error )
0541  {
0542      _logger( "KEYSERVER: could not contact database - fatal." )
              if ENABLED_LOGGING;

0543      die "keyserver: " . $connection->get_error_message . ".\n";
0544  }

0545  # Check to see if an entry exists in the database.  Start by assuming
0546  # the worst, that is: there is no PK- in database.

0547  my $pkplus_in_db = FALSE;

0548  my $query = 'select ip_address ' .
0549              'from publics where ' .
0550              '( ip_address = "' . LOCALHOST . '" and ' .
0551              'protocol_port = "' . RESPONDER_PPORT . '" )';

0552  $connection->query( $query );

0553  my $iterator = $connection->create_record_iterator;
0554  my $rec = $iterator->each;

0555  # The $rec scalar will reference an array if an entry was found in the
        database.

0556  if ( ref( $rec ) eq 'ARRAY' )
0557  {
0558      $pkplus_in_db = TRUE;
0559  }

0560  if ( !$pkplus_in_db )
0561  {
0562      # We need to (re)generate the PK-/PK+ pairing, update the database with
0563      # the PK+ and store the PK- in a disk-file.

0564      my $rsa = new Crypt::RSA;

0565      print "Generating a public/private key-pairing for this keyserver.  "
              if ENABLED_PRINTS;
0566      print "Please wait ... \n" if ENABLED_PRINTS;

0567      my $ksf = LOCALHOST . '.' . RESPONDER_PPORT;

0568      my ( $public, $private ) =
0569              $rsa->keygen(
0570                  Identity  => 'Scooby Key Server',
0571                  Size      => KEY_SIZE,
0572                  Password  => KEYSRV_PASSWD,
0573                  Filename  => $ksf,
0574                  Verbosity => FALSE
0575              );

0576      print "Generated.  Keyserver starting ... \n" if ENABLED_PRINTS;

0577      # The PK+ and PK- now exist in the "LOCALHOST.RESPONDER_PPORT.public"
0578      # and "LOCALHOST.RESPONDER_PPORT.private" disk-files.  So, add the PK+
0579      # to the 'SCOOBY.publics' table.

0580      open KEYFILE, "$ksf.public"
0581          or die "keyserver: The public KEYFILE does not exist: $!.\n";
```

```
0582      my @entire_keyfile = <KEYFILE>;

0583      close KEYFILE;

0584      # The assumption here is that the entry does NOT exist in the database,
0585      # so we use an INSERT as opposed to an UPDATE statement.

0586      $query = 'insert into publics ' .
0587               '( ip_address, protocol_port, public_key ) values  ' .
0588               '( "' . LOCALHOST . '", "' . RESPONDER_PPORT . '", ' .
0589               "\"@entire_keyfile\" )";

0590      $connection->query( $query );  # We (naively) assume success.
0591  }
0592  else
0593  {
0594      print "Using the existing public/private key-pairing for this
                keyserver.\n" if ENABLED_PRINTS;
0595      print "Keyserver starting ... \n" if ENABLED_PRINTS;
0596  }

0597  $connection->close;

0598  # Create a sub-process to handle the monitoring web server.

0599  my $http_pid = fork;

0600  if ( !defined( $http_pid ) )
0601  {
0602      _logger( "KEYSERVER: unable to create HTTP service." )
              if ENABLED_LOGGING;

0603      die "keyserver: unable to create HTTP subprocesses: $!.\n";
0604  }

0605  if ( $http_pid == FALSE )
0606  {
0607      _logger( "KEYSERVER: starting the HTTP service." ) if ENABLED_LOGGING;

0608      _start_web_service if ENABLED_LOGGING;

0609       exit 0;  # Which will execute if ENABLED_LOGGING is false.
0610  }
0611  else
0612  {
0613      # With the PK- and PK+ in place, we can now create the Responder and
0614      # Registration services by forking a child process.

0615      my $pid = fork;

0616      if ( !defined( $pid ) )
0617      {
0618          _logger( "KEYSERVER: unable to create subprocesses." )
                  if ENABLED_LOGGING;

0619          die "keyserver: unable to create initial subprocesses: $!.\n";
0620      }

0621      if ( $pid == FALSE )
0622      {
0623          # This is the child process executing.
```

```
0624          # This next call is NEVER returned from.

0625          _start_registration_service;
0626      }
0627      else
0628      {
0629          # This is the parent process executing.
0630          # This next call is NEVER returned from.

0631          _start_responder_service;
0632      }
0633  }

0634  ##############################################################################
0635  # Documentation starts here.
0636  ##############################################################################

0637  =head1 NAME

0638  keyserver - an RSA-based public keyserver for use with B<Devel::Scooby>
      (which includes HTTP monitoring facility at port 8080).

0639  =head1 VERSION

0640  1.04

0641  =head1 SYNOPSIS

0642  Create a ".keyserverrc" configuration file (see FILES), set-up the required
      database (see ENVIRONMENT), then invoke the keyserver:

0643  =over 4

0644      ./keyserver

0645  =back

0646  =head1 DESCRIPTION

0647  This keyserver provides three services to clients that communicate with it.

0648  1. The "Responder Service" runs on port B<RESPONDER_PPORT> and listens for
      requests from clients.  These take the form of an IP address in
      dotted-decimal notation, followed by a protocol port number.  The IP
      address/port-number are looked-up in the SCOOBY.publics table (see
      ENVIRONMENT), and - if found - the associated public key is extracted from
      the table and signed using this keyserver's private key.  Both the
      signature and the public key are then sent to the client.

0649  If the lookup fails, the strings "NOSIG" followed by "NOTFOUND" are returned
      to the client.

0650  If the IP address is LOCALHOST (which defaults to 127.0.0.1) and the
      protocol port number is RESPONDER_PPORT (which defaults to 30001), then
      this program returns the string "SELFSIG" followed by an UNSIGNED copy of
      this keyserver's public key.  In this way, a client can retrieve the public
      key to use when verifying signatures.

0651  2. The "Registration Service" runs on port B<REGISTRATION_PPORT> and listens
      for connections from clients.  When on arrives, it is immediately followed
      by a protocol port number, then a public key.  This key is added to the
      SCOOBY.publics table (see ENVIRONMENT) together with the clients IP address
```

```
      in dotted-decimal notation and the protocol port number.  For obvious
      reasons, the received public key is NOT signed by the client.

0652  Note that changing the defined constant values for B<REGISTRATION_PPORT>
      and B<RESPONDER_PPORT> from their defaults will require source code changes
      to programs that interact with this keyserver (which includes the
      B<Devel::Scooby>, B<Mobile::Executive> and B<Mobile::Location> modules).
      So, don't change these constant values unless you really have to.

0653  3. The "HTTP-based Monitoring Service" runs on port HTTP_PORT (which
      defaults to 8080), and provides a mechanism to remotely check the status
      of the keyserver via the world-wide-web.  The LOGFILE can be viewed and
      (optionally) reset via the web-based interface.  Resetting the LOGFILE
      results in an archived copy of the LOGFILE-to-date being created on the
      keyserver's local storage.

0654  =head1 ENVIRONMENT

0655  It is assumed that the MySQL RDBMS is executing on the same machine as this
      keyserver.  Here's a quick list of MySQL-specific instructions for creating
      a database and table required to support this program:

0656  =over 4

0657      mysql -u root -p

0658      mysql> create database SCOOBY;
0659      mysql> use mysql;
0660      mysql> grant all on SCOOBY.* to perlagent identified by 'passwordhere';
0661      mysql> quit

0662      mysql -u perlagent -p SCOOBY < create_publics.sql

0663  =back

0664  If you use a different user-id/password combo to that shown above, be sure
      to change the two constants defined at the start of the source code
      (KEYDB_USER and KEYDB_PASS).

0665  where the B<create_publics.sql> disk-file contains:

0666  =over 4

0667      create table publics
0668      (
0669          ip_address       varchar (16)  not null,
0670          protocol_port    varchar (6)   not null,
0671          public_key       text          not null
0672      )

0673  =back

0674  =head1 FILES

0675  A configuration file, called ".keyserverrc", needs to exist in the same
      directory as this keyserver.  Its contents detail the IP address and
      protocol port numbers that connections will be allowed from.  Typically, it
      will look something like this:

0676  =over 4

0677      127.0.0.1:*
```

```
0678      192.168.22.14:*

0679  =back

0680  which allows any connection (on any port) from both 127.0.0.1 and
      192.168.22.14.  Note that (at the moment), specifying a protocol port number
      in place of "*" has no effect.  Connection from all ports on the specified
      IP address are allowed.  This will change in a future release.

0681  When first executed, this keyserver creates two disk-files:

0682  =over 4

0683      "LOCALHOST.RESPONDER_PPORT.public", and
0684      "LOCALHOST.RESPONDER_PPORT.private".

0685  =back

0686  These contain this keyserver's RSA public and private keys, respectively.
      The public key is also added to the MySQL database.

0687  DO NOT remove these files from the directory that runs this keyserver.

0688  DO NOT edit these files, either.

0689  The keyserver also logs all communication with it (in a disk-file called
      "keyserver.log").  The contents of this log can be viewed (and archives of
      it created) using the "HTTP-based Monitoring Service" (see DESCRIPTION).

0690  =head1 FOUR IMPORTANT CONSTANTS

0691  Near the start of the keyserver's source code, four constants are defined
      as follows:

0692  =over 4

0693      use constant KEYSRV_PASSWD       => 'keyserver';
0694      use constant KEY_SIZE            => 1024;

0695      use constant ENABLED_LOGGING     => 1;
0696      use constant ENABLED_PRINTS      => 1;

0697  =back

0698  Change the first two constants to values of your choosing to set the
      password (KEYSRV_PASSWD) and the key size (KEY_SIZE) to use during the
      PK+/PK- generation.  Note: the larger the key size, the stronger the
      encryption, but, the slower this software will run.  The default value for
      KEY_SIZE should suffice for most situations.

0699  Set ENABLED_LOGGING to 0 switch off disk-based logging and the HTTP-based
      Monitoring Service.

0700  Set ENABLED_PRINTS to 0 to disable the the display of status messages on
      STDOUT.

0701  =head1 SEE ALSO

0702  The B<Devel::Scooby>, B<Mobile::Executive> and B<Mobile::Location> modules.

0703  The following CPAN modules are assumed to be installed: B<Net::MySQL> and
      B<Crypt::RSA>.  The HTTP server requires B<HTTP::Daemon> and
```

```
      B<HTTP::Status>, which are installed as part of the B<libwww-perl> library
      (also available on CPAN).

0704  The Scooby Website: B<http://glasnost.itcarlow.ie/~scooby/>.

0705  =head1 AUTHOR

0706  Paul Barry, Institute of Technology, Carlow in Ireland,
      B<paul.barry@itcarlow.ie>, B<http://glasnost.itcarlow.ie/~barryp/>.

0707  =head1 COPYRIGHT

0708  Copyright (c) 2003, Paul Barry.  All Rights Reserved.

0709  This module is free software.  It may be used, redistributed and/or modified
      under the same terms as Perl itself.
```

1.7 Executive.pm Source Code

```
0710  package Mobile::Executive;

0711  # Executive.pm - the mobile agent client support code.
0712  #
0713  # Author: Paul Barry, paul.barry@itcarlow.ie
0714  # Create: October 2002.
0715  # Update: April 2003 - version 1.x series supports relocation.
0716  #         May 2003 - version 2.x adds support for authentication and
0717  #             encryption using Crypt::RSA.

0718  require Exporter;

0719  our $VERSION      = 2.03;

0720  our @ISA          = qw( Exporter );

0721  # We export all the symbols declared in this module by default.

0722  our @EXPORT       = qw(
0723                           relocate
0724                           $absolute_fn
0725                           $public_key
0726                           $private_key
0727                       );

0728  our @EXPORT_OK    = qw(
0729                       );
0730
0731  our %EXPORT_TAGS = (
0732                     );
0733
0734  use constant KEY_ID    => 'Mobile::Executive ID';
0735  use constant KEY_SIZE  => 1024;
0736  use constant KEY_PASS  => 'Mobile::Executive PASS';

0737  use constant TRUE      => 1;
0738  use constant FALSE     => 0;


0739  BEGIN {

0740      # This BEGIN block is executed as soon as the module is "used".
0741      # We determine the absolute path and filename of the program using
0742      # this module.  This is important, as the Devel::Scooby.pm module needs
0743      # this information during a relocate.  Note the use of 'our'.
0744      # We also generate a PK+ and PK- for "users" of this module.

0745      use Crypt::RSA;  # Provides authentication and encryption services.
0746      use File::Spec;  # Provides filename and path services.

0747      our $absolute_fn = File::Spec->rel2abs( File::Spec->curdir ) . '/' . $0;

0748      my $rsa = new Crypt::RSA;

0749      our ( $public_key, $private_key ) =
0750              $rsa->keygen(
0751                  Identity  => KEY_ID . "$$" . "$0",
0752                  Size      => KEY_SIZE,
0753                  Password  => KEY_PASS . "$0" . "$$",
```

```
0754                 Verbosity => FALSE
0755             ) or die $rsa->errstr, "\n";
0756 }

0757 sub relocate {

0758     # The relocate subroutine.
0759     #
0760     # IN:  The IP name/address and protocol port number of a Location to
0761     #      relocate to.
0762     #
0763     # OUT: nothing.

0764     my $ip_address    = shift;
0765     my $protocol_port = shift;

0766     # Does nothing - just a place holder.  The Devel::Scooby module
0767     # runs its own relocate code as part of its "sub" invocation.  That is,
0768     # a call to this relocate results in the Devel::Scooby running its own
0769     # version of "relocate".

0770     return;
0771 }

0772 1;  # As it is required by Perl.

0773 ##############################################################################
0774 # Documentation starts here.
0775 ##############################################################################

0776 =pod

0777 =head1 NAME

0778 "Mobile::Executive" - used to signal the intention to relocate a Scooby
     mobile agent from the current Location to some other (possibly remote)
     Location.

0779 =head1 VERSION

0780 2.03 (version 1.0x never released).

0781 =head1 SYNOPSIS

0782 use Mobile::Executive;

0783    ...

0784 relocate( $remote_location, $remote_port );

0785 =head1 DESCRIPTION

0786 Part of the Scooby mobile agent machinery, the B<Mobile::Executive> module
     provides a means to signal the agents intention to relocate to another
     Location.  Typical usage is as shown in the B<SYNOPSIS> section above.
     Assuming an instance of B<Mobile::Location> is executing on
     B<$remote_location> at protocol port number B<$remote_port>, the agent
     stops executing on the current Location, relocates to the remote Location,
     then continues to execute from the statement immediately AFTER the
     B<relocate> statement.

0787 Note: a functioning keyserver is required.
```

```
0788  =head1 Overview

0789  The only subroutine provided to programs that use this module is:

0790  =over 4

0791  relocate

0792  =back

0793  and it takes two parameters: a IP address (or name) of the remote Location,
      and the protocol port number that the Location is listening on.

0794  =head1 Internal methods/subroutines

0795  A Perl B<BEGIN> block determines the absolute path to the mobile agents
      source code file, and puts it into the B<$absolute_fn> scalar (which is
      automatically exported).  This block also generates a PK+/PK- pairing (in
      B<$public_key> and B<$private_key>) and exports both values (as they are
      used by B<Devel::Scooby>).

0796  =head1 RULES FOR WRITING MOBILE AGENTS

0797  There used to be loads, but now there is only one.  Read the
      B<Scooby Guide>, available on-line at:
      B<http://glasnost.itcarlow.ie/~scooby/guide.html>.

0798  =head1 SEE ALSO

0799  The B<Mobile::Location> class (for creating Locations), and the
      B<Devel::Scooby> module (for running mobile agents).

0800  The Scooby Website: B<http://glasnost.itcarlow.ie/~scooby/>.

0801  =head1 AUTHOR

0802  Paul Barry, Institute of Technology, Carlow in Ireland,
      B<paul.barry@itcarlow.ie>, B<http://glasnost.itcarlow.ie/~barryp/>.

0803  =head1 COPYRIGHT

0804  Copyright (c) 2003, Paul Barry.  All Rights Reserved.

0805  This module is free software.  It may be used, redistributed and/or
      modified under the same terms as Perl itself.
```

1.8 Scooby.pm Source Code

```
0806  package Devel::Scooby;

0807  # Scooby.pm - a relocation mechanism for use with the Mobile::Location
0808  #             and Mobile::Executive modules.
0809  #
0810  # Author: Paul Barry, paul.barry@itcarlow.ie
0811  # Create: October 2002.
0812  # Update: April/May 2003 - Version 4.x series.
0813  #
0814  # Notes:  This code takes advantage of the CPAN modules
0815  #         PadWalker and Storable (with a little help from the
0816  #         Data::Dumper module when it comes to Objects).  The Crypt::RSA
0817  #         module provides PK+/PK- support.
0818  #
0819  #         Version 1.x supported relocating simple Perl code.
0820  #         Version 2.x supported relocating SCALARs, ARRAYs, and
0821  #             HASHes and references to same.
0822  #         Version 3.x supported relocating Perl OO objects.  Note
0823  #             that this will only occur after Scooby has contacted
0824  #             the receiving Location and determined that any
0825  #             required classes exist on the remote Perl system.
0826  #         Version 4.x supports authenticated relocation using Crypt::RSA,
0827  #             as well as encryption of the mobile agent source code.
0828  #

0829  our $VERSION = 4.12;

0830  # The "constant.pm" module does not want to work with the debugger
0831  # mechanism, so "our" variables are used instead.

0832  our $SCOOBY_CONFIG_FILE   = "$ENV{'HOME'}/.scoobyrc";

0833  our $SIGNATURE_DELIMITER  = "\n--end-sig--\n";

0834  our $ALARM_WAIT           = 30;
0835  our $LOCALHOST            = '127.0.0.1';

0836  our $RESPONDER_PPORT      = '30001';
0837  our $REGISTER_PPORT       = '30002';

0838  our $MAX_RECV_LEN         = 65536;

0839  our $TRUE                 = 1;
0840  our $FALSE                = 0;

0841  ##############################################################################
0842  # The Scooby Debugger starts here.
0843  ##############################################################################

0844  {
0845      package DB;  # Remember: Scooby is a DEBUGGER.
0846
0847      our ( $package, $file, $line );  # XXXXX: Note these are 'global'.

0848      sub DB {

0849          # Called for every line in the program that can be breakpointed.
0850          #
0851          # IN:  nothing.
```

```
0852          #
0853          # OUT: nothing.

0854          ( $package, $file, $line ) = caller;  # XXXXX: Writing to globals!
0855      }

0856      sub sub {

0857          # Called before every subroutine call in the program.
0858          #
0859          # IN:  nothing.  Although "$sub" is set to the name of the
0860          #      subroutine that was just called (thanks to Perl's debugging
0861          #      mechanisms).
0862          #
0863          # OUT: nothing.

0864          if ( $sub =~ /^Mobile::Executive::relocate$/ )
0865          {
0866              use Socket;                      # Functional interface to
                                                     Socket API.
0867              use Storable qw( freeze thaw );  # Provides a persistence
                                                     mechanism.
0868              use PadWalker qw( peek_my );     # Provides access to all
                                                     lexically scoped variables.

0869              use Crypt::RSA;                  # Provides authentication and
0870                                               # encryption services.

0871              my $remote = shift;

0872              # Next two lines turn the IP name into a dotted-decimal.

0873              my $tmp = gethostbyname( $remote ) or inet_aton( $remote );
0874              $remote = inet_ntoa( $tmp );

0875              my $remote_port = shift;

0876              my $filename_mem = $file;
0877              my $linenum_mem = ( $line + 1 );

0878              my $stringified;
0879
0880              # We first determine the list of lexicals in the caller.

0881              my $them = peek_my( 0 );

0882              # Then we turn the list of lexicals into "Storable" output.

0883              my $str = freeze( \%{ $them } );

0884              # Then we turn the thawed output back into Perl code.  This
0885              # code is referred to as the "lexical init" code.

0886              $stringified = _storable_decode(
0887                                                  $remote,
0888                                                  $remote_port,
0889                                                  thaw( $str )
0890                                              );

0891              # Determine the KEYSERVER address from the .scoobyrc file.

0892              open KEYFILE, "$SCOOBY_CONFIG_FILE"
```

```
0893                or die "Scooby: unable to access ~/.scoobyrc. Does it
                          exist?\n";

0894            my $keyline = <KEYFILE>;

0895            close KEYFILE;

0896            # Note: format of 'rc' file is very strict.  No spaces!

0897            $keyline =~ /^KEYSERVER=(.+)/;

0898            my $key_server = $1;

0899            # Now that we know the address of the key server, we can
0900            # request the PK+ of the key server and the next location.

0901            _get_store_pkplus( $key_server, $LOCALHOST, $RESPONDER_PPORT );
0902            _get_store_pkplus( $key_server, $remote, $remote_port );

0903            open RELOCATE_FILE, "$Mobile::Executive::absolute_fn"
0904                or die "Scooby: Unable to open file for relocation: $!.\n";

0905            # Dump the current state of the agent to a temporary disk-file
0906            # so that we can encrypt it with the next Location's PK+.

0907            my $tmp_filename = "$0.$$.temporary.tmp";

0908            open TMP_FILE, ">$tmp_filename"
0909                or die "Scooby: could not write to temporary encryption
                          file: $!.\n";

0910            my $line_count = 0;

0911            # Write the agent's source code one line at a time to the
                      temporary file.

0912            while ( my $line2send = <RELOCATE_FILE> )
0913            {
0914                ++$line_count;
0915                print TMP_FILE $line2send;

0916                # Check to see if we need to insert the "lexical init" code.

0917                if ( $line_count == ($linenum_mem-1) )
0918                {
0919                    print TMP_FILE $stringified if defined( $stringified );
0920                }
0921            }

0922            close RELOCATE_FILE;

0923            close TMP_FILE;

0924            # The agent source code (which has mutated) is now in
                      "$tmp_filename".

0925            open TOENCRYPT_FILE, "$tmp_filename"
0926                or die "Scooby: temporary encryption file could not be
                          opened: $!.\n";

0927            my @entire_toencrypt = <TOENCRYPT_FILE>;
```

```
0928            close TOENCRYPT_FILE;

0929            # We are now done with the temporary file, so we can remove it
0930            # from the local storage.

0931            unlink $tmp_filename;

0932            my $message = "@entire_toencrypt\n";
0933            my $public_key_filename = "$remote.$remote_port.public";

0934            my $public_key = new Crypt::RSA::Key::Public(
0935                                 Filename => $public_key_filename
0936                             );

0937            my $rsa = new Crypt::RSA;

0938            # Encrypt the mutated agent using the PK+ of the next Location.

0939            my $cyphertext = $rsa->encrypt(
0940                                 Message   => $message,
0941                                 Key       => $public_key,
0942                                 Armour    => $TRUE
0943                            ) or die $rsa->errstr, "\n";

0944            # Use the PK- of this Mobile::Executive invocation to
0945            # sign the encrypted mobile agent.

0946            my $cypher_signature = $rsa->sign(
0947                                       Message  => $cyphertext,
0948                                       Key      =>
                                                $Mobile::Executive::private_key,
0949                                       Armour   => $TRUE
0950                                   ) or die $rsa->errstr, "\n";

0951            # Networking code to send agent to the server starts here.

0952            my $trans_serv  = getprotobyname( 'tcp' );
0953            my $remote_host = gethostbyname( $remote ) or
                      inet_aton( $remote );
0954            my $destination = sockaddr_in( $remote_port, $remote_host );

0955            socket( TCP_SOCK, PF_INET, SOCK_STREAM, $trans_serv )
0956                or die "Scooby: socket creation failed: $!.\n";

0957            connect( TCP_SOCK, $destination )
0958                or die "Scooby: connect to remote system failed: $!.\n";

0959            # Turn on auto-flushing.

0960            my $previous = select TCP_SOCK;
0961            $| = 1;
0962            select $previous;

0963            # Send the filename of the agent to the remote Location.

0964            print TCP_SOCK $filename_mem . "\n";

0965            # Send the line# for the next executable line to the Location.

0966            print TCP_SOCK $linenum_mem . "\n";

0967            # We need to work out the port that this client is using
```

```
                    "locally".
0968            # The Location will use this protocol port number to query the
0969            # keyserver for the just-about-to-be-sent public key.

0970            my ( $local_pport, $local_ip ) =
                    sockaddr_in( getsockname( TCP_SOCK ) );

0971            # Prior to sending the signature and cyphertext to the next
0972            # Location, we need to update the keyserver with the appropriate
0973            # PK+ so that the next Location can verify the signature.  We
0974            # write the PK+ to a disk-file, then read it back in, as this is
0975            # the format that the keyserver expects to receive it in.

0976            $Mobile::Executive::public_key->write(
0977                Filename => "$0.$$.$local_pport.public"
0978            );

0979            open LOCAL_KEYFILE, "$0.$$.$local_pport.public"
0980                or die "Scooby: the local public key file does not
                           exist: $!.\n";

0981            my @entire_local_file = <LOCAL_KEYFILE>;

0982            close LOCAL_KEYFILE;

0983            # We have no further need for the public key file, so remove it.

0984            unlink "$0.$$.$local_pport.public";

0985            # Send the "local" protocol port number and PK+ to the
                    keyserver.

0986            my $keysock_obj = IO::Socket::INET->new( PeerAddr =>
                                                             $key_server,
0987                                                     PeerPort =>
                                                             $REGISTER_PPORT,
0988                                                     Proto    => 'tcp' );

0989            if ( !defined( $keysock_obj ) )
0990            {
0991                die "Scooby: could not create socket object to key
                           server: $!.\n";
0992            }

0993            print $keysock_obj "$local_pport\n";

0994            print $keysock_obj @entire_local_file;

0995            $keysock_obj->close;

0996            # ACK that the just inserted PK+ is in the keyserver.

0997            _wait_for_pkplus_confirm( $key_server, inet_ntoa( $local_ip ),
                    $local_pport );

0998            # Send the signature to the next Location.

0999            print TCP_SOCK "$cypher_signature" . $SIGNATURE_DELIMITER;

1000            # Send the encoded cyphertext to the next Location.

1001            print TCP_SOCK $cyphertext;
```

```
1002             close TCP_SOCK
1003                 or warn "Scooby: close failed: $!.\n";

1004             exit;  # We are done on this Location, having just relocated
1005                    # to another.  This is why we "exit" at this time.
1006         }

1007         # Call the original subroutine with parameters (if there was any).
1008         # We only get to here if there's no request for relocation.

1009         if ( defined @_ )
1010         {
1011             &$sub( @_ );
1012         }
1013         else
1014         {
1015             &$sub;
1016         }
1017     }

1018 ##############################################################################
1019 # Scooby support routines follow.
1020 ##############################################################################

1021     sub _wait_for_pkplus_confirm {

1022         # Contacts the key server and requests the PK+ for a specified
1023         # IP address/port combo.  Keeps asking for the PK+ until such time
1024         # as the PK+ is ACKed by the key server.
1025         #
1026         # IN:  The IP name/address of the key server.
1027         #      The IP address to use when requesting a PK+ from key server.
1028         #      The protocol port to use when requesting a PK+.
1029         #
1030         # OUT: nothing.
1031
1032         use IO::Socket;  # Provides OO interface to Socket API.
1033
1034         my $server = shift;
1035         my $lookup = shift;
1036         my $port   = shift;
1037
1038         my $sig_ack = $FALSE;

1039         while ( $sig_ack == $FALSE )
1040         {
1041             # Opens a socket object to the keyserver.

1042             my $key_sock = IO::Socket::INET->new(
1043                                                  PeerAddr => $server,
1044                                                  PeerPort =>
                                                         $RESPONDER_PPORT,
1045                                                  Proto    => 'tcp'
1046                                                );

1047             if ( !defined( $key_sock ) )
1048             {
1049                 die "Scooby: could not create key server socket object:
                           $!.\n";
1050             }
1051
```

```
1052            # Send the lookup details to the keyserver.

1053            print $key_sock "$lookup\n";
1054            print $key_sock $port;
1055
1056            # We are done writing, so half close the socket.

1057            $key_sock->shutdown( 1 );
1058
1059            my $data = '';

1060            # Read the entire response from the keyserver.
1061
1062            while ( my $chunk = <$key_sock> )
1063            {
1064                $data = $data . $chunk;
1065            }
1066
1067            $key_sock->close;
1068
1069            # This splits the signature and data on the SIGNATURE_DELIMITER
1070            # pattern as used by the keyserver.

1071            ( my $key_sig, $data ) = split /\n--end-sig--\n/, $data;
1072
1073            if ( $key_sig eq "NOSIG" )
1074            {
1075                $sig_ack = $FALSE;
1076            }
1077            else
1078            {
1079                $sig_ack = $TRUE;
1080            }
1081        }
1082    }

1083    sub _get_store_pkplus {

1084        # Contacts the key server and requests the PK+ for a specified
1085        # IP address/port combo. Stores the PK+ in the named disk-file.
1086        #
1087        # IN:  The IP name/address of the key server.
1088        #      The IP address to use when requesting a PK+ from key server.
1089        #      The protocol port to use when requesting a PK+.
1090        #
1091        # OUT: nothing.
1092        #
1093        # This code is an extension of the "_wait_for_pkplus_confirm" code.
1094
1095        use Crypt::RSA;  # Provides authentication and encryption services.
1096        use IO::Socket;  # Provides OO interface to Socket API.
1097
1098        my $server = shift;
1099        my $lookup = shift;
1100        my $port   = shift;
1101
1102        my $key_sock = IO::Socket::INET->new(
1103                                              PeerAddr => $server,
1104                                              PeerPort => $RESPONDER_PPORT,
1105                                              Proto    => 'tcp'
1106                                            );
```

```
1107        if ( !defined( $key_sock ) )
1108        {
1109            die "Scooby: could not create key server socket object: $!.\n";
1110        }
1111
1112        print $key_sock "$lookup\n";
1113        print $key_sock $port;
1114
1115        # We are done writing, so half close the socket.

1116        $key_sock->shutdown( 1 );
1117
1118        my $data = '';
1119
1120        while ( my $chunk = <$key_sock> )
1121        {
1122            $data = $data . $chunk;
1123        }
1124
1125        $key_sock->close;
1126
1127        # This splits the signature and data on the SIGNATURE_DELIMITER
1128        # pattern as used by the keyserver.

1129        ( my $key_sig, $data ) = split /\n--end-sig--\n/, $data;

1130        if ( $key_sig eq "NOSIG" )
1131        {
1132            die "Scooby: no signature found: aborting.\n";
1133        }
1134        elsif ( $key_sig eq "SELFSIG" )
1135        {
1136            my $lf = "$lookup.$port.public";  # Location PK+ filename.

1137            open KEYFILE, ">$lf"
1138                or die "Scooby: could not create key file: $!.\n";
1139
1140            print KEYFILE $data;
1141
1142            close KEYFILE;
1143        }
1144        else
1145        {
1146            my $ksf = "$LOCALHOST.$RESPONDER_PPORT.public";

1147            my $key_server_pkplus = new Crypt::RSA::Key::Public(
1148                                            Filename => $ksf
1149                                        );
1150
1151            my $rsa = new Crypt::RSA;
1152
1153            my $verify = $rsa->verify(
1154                                         Message    => $data,
1155                                         Signature  => "$key_sig",
1156                                         Key        => $key_server_pkplus,
1157                                         Armour     => $TRUE
1158                                       );
1159
1160            if ( !$verify )
1161            {
1162                die "Scooby: signature for next location does not verify:
                          aborting.\n";
```

```
1163            }
1164            else
1165            {
1166                open KEYFILE, ">$lookup.$port.public"
1167                    or die "Scooby: could not create key file: $!.\n";
1168
1169                print KEYFILE $data;
1170
1171                close KEYFILE;
1172            }
1173        }
1174    }
1175
1176    sub _check_modules_on_remote {

1177        # Contacts the remote Location, sends the list of required modules,
1178        # waits for a response, then returns it to the caller.
1179        #
1180        # IN:   The IP name (or address) of the remote Location.
1181        #       The protocol port number of the remote Location.
1182        #       The list of modules to look for.
1183        #
1184        # OUT:  The message received from the server.
1185
1186        my $remote         = shift;
1187        my $remote_port    = shift;
1188        my @tocheck        = @_;
1189
1190        use Socket;  # Functional interface to Socket API.
1191
1192        my $trans_serv  = getprotobyname( 'tcp' );
1193        my $remote_host = gethostbyname( $remote ) or inet_aton( $remote );
1194
1195        # Note: the server listens at Port+1.

1196        my $destination = sockaddr_in( $remote_port+1, $remote_host );
1197
1198        socket( CHECK_MOD_SOCK, PF_INET, SOCK_STREAM, $trans_serv )
1199            or die "Scooby: socket creation failed: $!.\n";
1200        my $con_ok = connect( CHECK_MOD_SOCK, $destination )
1201            or die "Scooby: connect to remote system failed: $!.\n";
1202
1203        # Send the list of modules to check.

1204        send( CHECK_MOD_SOCK, join( ' ', @tocheck ), 0 )
1205            or warn "Scooby: problem with send: $!.\n";
1206
1207        shutdown( CHECK_MOD_SOCK, 1 );  # Close the socket for writing.
1208
1209        my $remote_response = '';
1210
1211        # Add a signal handler to execute when the alarm sounds
              (or expires).

1212        $SIG{ALRM} = sub { die "no remote module check\n"; };
1213
1214        alarm( $ALARM_WAIT );
1215
1216        # We wait for up to ALARM_WAIT seconds for a response from the
              Location.

1217        eval {
```

```
1218            recv( CHECK_MOD_SOCK, $remote_response, $MAX_RECV_LEN, 0 );

1219            alarm( 0 );  # Cancel the alarm, we do not need it now.
1220        };
1221
1222        close CHECK_MOD_SOCK
1223            or warn "Scooby: close failed: $!.\n";
1224
1225        # Process the timeout if it happened.  Die if we see some message
1226        # other than the one we expect.

1227        if ( $@ )
1228        {
1229            die "Scooby: $@\n" unless $@ =~ /no remote module check/;
1230
1231            warn "Scooby: not able to check existence of remote modules.\n";
1232        }
1233
1234        return $remote_response;
1235    }
1236
1237    sub _storable_decode {

1238        # Called immediately after the lexical variables are stringified
1239        # in order to return the "Storable" output to its original form.
1240        #
1241        # IN:   The IP name (or address) of the remote Location.
1242        #       The protocol port number of the remote Location.
1243        #       The "thawed" output from the Storable::thaw method.
1244        #
1245        # OUT:  The stringified representation of the Perl code that can be
1246        #       executed to reinitialize the relocated variables.
1247        #
1248        # NOTE: This code also checks to see if any required modules exist
1249        #       on the remote Location.  It will "die" if some are missing.
1250
1251        my $remote      = shift;
1252        my $remote_port = shift;
1253        my $thawed      = shift;
1254
1255        my %for_refs;
1256        my $stringified = '';
1257        my @required_classes = ();
1258
1259        # The lexicals are processed TWICE, as it is not possible to
1260        # handle REFerences with a single pass over "$thawed".
1261
1262        # Process the lexicals once, for SCALARs, ARRAYs and HASHes.
1263        #
1264        # Note: we need to remember the 'memory address' of each variable,
1265        # so we check them against any REFerences in the second pass.
1266        #
1267        # The generated code is indented by four spaces.

1268        while ( my ( $name, $value ) = each ( %{ $thawed } ) )
1269        {
1270            if ( ref( $value ) eq 'SCALAR' )
1271            {
1272                $for_refs{ $value } = $name;

1273                # We do NOT want to enclose SCALAR numbers in quotes!
```

```
1274                 if ( $$value =~ /[^0123456789.]+/ )
1275                 {
1276                     $stringified .= "    $name = \"$$value\";\n";
1277                 }
1278                 else
1279                 {
1280                     $stringified .= "    $name = $$value;\n";
1281                 }
1282             }

1283             if ( ref( $value ) eq 'ARRAY' )
1284             {
1285                 $for_refs{ $value } = $name;
1286                 $stringified .= "    $name = qw( @$value );\n";
1287             }

1288             if ( ref( $value ) eq 'HASH' )
1289             {
1290                 $for_refs{ $value } = $name;
1291                 $stringified .= "    $name = (\n";
1292                 while ( my ( $h_name, $h_value ) = each ( %{ $value } ) )
1293                 {
1294                     $stringified .= "        \"$h_name\" => \"$h_value\",\n"
1295                 }
1296                 $stringified .= "    );\n";
1297             }
1298         }
1299
1300         # Second pass: process the lexicals again, this time for REFs.

1301         while ( my ( $name, $value ) = each ( %{ $thawed } ) )
1302         {

1303             # Deal with references to Perl OO objects.

1304             if ( ref( $value ) eq 'REF' && !defined( $for_refs{ $$value } ) )
1305             {
1306                 push @required_classes, ref( $$value );
1307
1308                 use Data::Dumper;
1309
1310                 my $string = Dumper( $value );
1311
1312                 # Make sure the appropriate Class is used.

1313                 $stringified .= "    use " . ref( $$value ) . ";\n\n";
1314
1315                 # Replace Data::Dumper's generated $VARn with correct name.

1316                 $string =~ s/^\$VAR\d+ = \\//;
1317
1318                 # Add the code to bless the object to the stringified code.

1319                 $stringified .= "    $name = $string\n";
1320             }
1321
1322             # Deal with references to SCALARs, ARRAYs and HASHes.

1323             if ( ref( $value ) eq 'REF' && defined( $for_refs{ $$value } ) )
1324             {
1325                 $stringified .= "    $name = \\$for_refs{ $$value };\n";
1326             }
```

```
1327          }
1328
1329          # Check to see if any required modules exist on the remote Location.
1330          # The list provided is calculated as a result of processing any
1331          # references to object instances.

1332          if ( @required_classes )
1333          {
1334              my $message = _check_modules_on_remote(
1335                                                      $remote,
1336                                                      $remote_port,
1337                                                      @required_classes
1338                                                  );
1339
1340              if ( $message =~ /^NOK/ )
1341              {
1342                  $message =~ s/^NOK: //;

1343                  die "Required modules missing on remote: $message.\n";
1344              }
1345              elsif ( $message !~ /^OK/ )
1346              {
1347                  warn "Something strange has happened: $message.\n";

1348                  die "Is the remote Location ready?\n";
1349              }
1350          }
1351
1352          # Assuming we haven't died, return the Perl code to the caller.

1353          return $stringified;
1354      }

1355  } # End of DB package.

1356  1;  # Evaluate true as last statement of this package (required by Perl).

1357  ##############################################################################
1358  # Documentation starts here.
1359  ##############################################################################

1360  =pod

1361  =head1 NAME

1362  "Scooby" - the internal machinery that works with B<Mobile::Location> and
      B<Mobile::Executive> to provide a mobile agent execution and location
      environment for the Perl Programming Language.

1363  =head1 VERSION

1364  4.0x (versions 1.x and 2.x were never released; version 3.x did not support
      encryption and authentication).

1365  =head1 SYNOPSIS

1366  perl -d:Scooby mobile_agent

1367  =head1 DESCRIPTION

1368  This is an internal module that is not designed to be "used" directly by a
      program.  Assuming a mobile agent called B<multiwho> exists (that "uses"
```

```
      the B<Mobile::Executive> module), this module can be used to execute it,
      as follows:

1369  =over 4

1370  perl -d:Scooby multiwho

1371  =back

1372  The B<-d> switch to C<perl> invokes Scooby as a debugger.  Unlike a
      traditional debugger that expects to interact with a human, Scooby runs
      automatically.  It NEVER interacts with a human, it interacts with the
      mobile agent machinery.

1373  Scooby can be used to relocate Perl source code which contains the
      following:

1374  =over 4

1375  SCALARs (both numbers and strings).

1376  An ARRAY of SCALARs (known as a simple ARRAY).

1377  A HASH of SCALARs (known as a simple HASH).

1378  References to SCALARs.

1379  References to a simple ARRAY.

1380  References to a simple HASH.

1381  Objects.

1382  References to objects are B<not> supported and are in no way guaranteed to
      behave the way you expect them to after relocation (even though they do
      relocate).

1383  The relocation of more complex data structures is B<not> supported at this
      time (refer to the TO DO LIST section, below).

1384  =back

1385  =head1 Internal methods/subroutines

1386  =over 4

1387  B<DB::DB> - called for every executable statement contained in the mobile
      agent source code file.

1388  B<DB::sub> - called for every subroutine call contained in the mobile agent
      source code file.

1389  B<_DB::storable_decode> - takes the stringified output from B<Storable>'s
      B<thaw> subroutine and turns it back into Perl code (with a little help
      from Data::Dumper for objects).

1390  B<DB::_check_modules_on_remote> - checks to see if a list of modules/classes
      "used" within the mobile agent actually exist on the remote Location's Perl
      system.

1391  B<DB::_get_store_pkplus> - contacts the key server and requests a PK+, then
      stores the PK+ in a named disk-file.
```

```
1392  B<DB::_wait_for_pkplus_confirm> - repeatedly contacts the key server until
      requested PK+ is returned (i.e., ACKed).

1393  =back

1394  =head1 ENVIRONMENT

1395  This module must be installed in your Perl system's B<Devel/> directory.
      This module will only work on an operating system that supports the Perl
      modules listed in the SEE ALSO section, below.  (To date, I've only tested
      it on various Linux distributions).

1396  =head1 TO DO LIST

1397  Loads.  The biggest item on the list would be to enhance Scooby to allow it
      to handle more complex data structures, such as ARRAYs of HASHes and HASHes
      of ARRAYs, etc., etc.

1398  My initial plan was to allow for the automatic relocation of open
      disk-files.  However, on reflection, I decided not to do this at this time,
      but may return to the idea at some stage in the future.

1399  The current implementation checks to see if "used" classes are available on
      the next Location before attempting relocation, but does not check to see
      if "used" modules are available.  It would be nice if it did.

1400  It would also be nice to incorporate an updated B<Class::Tom> (by James
      Duncan) to handle the relocation of objects to a Location without the need
      to have the module exist on the remote Location.  On my system (Linux), the
      most recent B<Class::Tom> generates compile/run-time errors.

1401  =head1 SEE ALSO

1402  The B<Mobile::Executive> module and the B<Mobile::Location> class.
      Internally, this module uses the following CPAN modules: B<PadWalker> and
      B<Storable>, in addition to the standard B<Data::Dumper> module.  The
      B<Crypt::RSA> modules provides encryption and authentication services.

1403  The Scooby Website: B<http://glasnost.itcarlow.ie/~scooby/>.

1404  =head1 AUTHOR

1405  Paul Barry, Institute of Technology, Carlow in Ireland,
      B<paul.barry@itcarlow.ie>, B<http://glasnost.itcarlow.ie/~barryp/>.

1406  =head1 COPYRIGHT

1407  Copyright (c) 2003, Paul Barry.  All Rights Reserved.

1408  This module is free software.  It may be used, redistributed and/or
      modified under the same terms as Perl itself.
```

1.9 Location.pm Source Code

```
1409  package Mobile::Location;

1410  # Location.pm - the mobile agent environment location class.
1411  #
1412  # Author: Paul Barry, paul.barry@itcarlow.ie
1413  # Create: March 2003.
1414  # Update: April 2003  - changed to IO::Socket for agent receipt/processing
1415  #                       due to "fork" strangeness on regular sockets.
1416  #         May 2003    - added support for authentication and encryption.
1417  #                     - added the web-based monitoring service.
1418  #
1419  # Notes:  Version 1.x - unsafe, totally trusting Locations (never released).
1420  #         Version 2.x - added support to the Location for executing mobile
1421  #                       agents within a restricted Opcode environment.
1422  #         Version 3.x - adds support for authentication and encryption.  This
1423  #                       code assumes that a functioning keyserver is running.
1424  #         Version 4.x - embeds a web-server to allow for remote monitoring
1425  #                       via the world-wide-web.

1426  use strict;

1427  use Crypt::RSA;       # Provides authentication and encryption services.
1428  use IO::Socket;       # OO interface to Socket API.
1429  use Socket;           # Procedural interface to Socket API.
1430  use Sys::Hostname;    # Provides means to determine name of current machine.
1431  use HTTP::Daemon;     # Provides a basic HTTP server.
1432  use HTTP::Status;     # Provides support for HTTP status messages.
1433  use POSIX 'WNOHANG';  # Provides support for POSIX signals.

1434  # Add a signal handler to process and deal with "zombies".

1435  $SIG{CHLD} = sub { while ( waitpid( -1, WNOHANG ) > 0 ) { }; };

1436  our $VERSION = 4.02;

1437  use constant TRUE               => 1;
1438  use constant FALSE              => 0;

1439  use constant RUN_LOCATION_DIR   => "Location";
1440  use constant KEY_SIZE           => 1024;

1441  use constant RESPONDER_PPORT    => '30001';
1442  use constant REGISTRATION_PPORT => '30002';

1443  use constant SCOOBY_CONFIG_FILE => "$ENV{'HOME'}/.scoobyrc";

1444  use constant HTML_DEFAULT_PAGE  => "index.html";
1445  use constant HTTP_PORT          => 8080;

1446  use constant LOGFILE            => 'location.log';

1447  use constant VISIT_SCOOBY       => 'Visit the <a href="http://glasnost
                                         .itcarlow.ie/~scooby/">Scooby Website</a>
                                         at IT Carlow.<p>';

1448  our $_PWD = ''; # This 'global' contains the current working directory
1449                  # for the Location instance determined during construction.

1450  ##############################################################################
```

```
1451  # The class constructor is in "new".
1452  ##############################################################################

1453  sub new {

1454      # The Mobile::Location constructor.
1455      #
1456      # IN:   Receives a series of optional name/value pairings.
1457      #           Port  - Protocol port value to accept connections from.
1458      #                   Default value for Port is '2001'.
1459      #           Debug - set to 1 for STDERR status messages.
1460      #                   Default value for Debug is 0 (off).
1461      #           Log   - set to 1 to enable logging of agents to disk.
1462      #                   Default value for Log is 0 (off).
1463      #           Ops   - a set of Opcodes or Opcode tags, which are
1464      #                   added to Scooby's ALLOWED ops when executing
1465      #                   mobile agents.
1466      #           Web   - set to 1 to enable the logging mechanism and the
1467      #                   creation of a HTTP-based Monitoring Service.  The
1468      #                   default is 1 (i.e., ON).
1469      #
1470      # OUT:  Returns a blessed reference to a Mobile::Location object.

1471      my ( $class, %arguments ) = @_;

1472      my $self = bless {}, $class;

1473      $self->{ Port }  = $arguments{ Port }  || 2001;
1474      $self->{ Debug } = $arguments{ Debug } || FALSE;
1475      $self->{ Log }   = $arguments{ Log }   || FALSE;
1476      $self->{ Ops }   = $arguments{ Ops }   || '';
1477      $self->{ Web }   = $arguments{ Web }   || TRUE;

1478      # Untaint the PATH by setting it to something really limited.

1479      $ENV{'PATH'} = "/bin:/usr/bin";

1480      # This next line is part of the standard Perl technique. See 'perlsec'.

1481      delete @ENV{ 'IFS', 'CDPATH', 'ENV', 'BASH_ENV' };
1482
1483      $_PWD = `pwd`; # XXXXXX: Writing to global!  This is tainted.
1484      $_PWD =~ /^([-\@\/\w_.]+)$/; # So, we untaint it, using a regex.
1485      $_PWD = $1;

1486      # Disallow if running this Location as 'root'.

1487      die "Location running as ROOT. This is NOT secure (nor allowed)!"
1488          unless $> and $^O ne 'VMS';

1489      # Work out and remember the IP address of the computer running this
               Location.

1490      my $host = gethostbyname( hostname ) or inet_aton( hostname );
1491      $self->{ Host } = inet_ntoa( $host );

1492      # Generate and remember a password to use with the PK- and PK+.

1493      $self->{ Password } = $0 . $$ . '_Location';

1494      # NOTE: A second server is spawned at this stage to handle any
1495      #       requests from an agent re: the availability of any
```

```
1496      #       required modules within the Perl system running this Location.
1497      #       See the _check_modules_on_remote subroutine from Devel::Scooby,
1498      #       as well as the _spawn_network_service and _check_for_modules
1499      #       subroutines, below.

1500      _spawn_network_service( $self->{ Port }+1 );

1501      # Create the HTTP-based Monitoring Service.

1502      $self->_spawn_web_monitoring_service;

1503      return $self;
1504  }


1505  ##############################################################################
1506  # Methods and support subroutines.
1507  ##############################################################################

1508  sub _logger {

1509      # This small routine quickly writes a message to the LOGFILE.  Note
1510      # that every line written to the LOGFILE is timestamped.
1511      #
1512      # IN:  a message to log.
1513      #
1514      # OUT: nothing.

1515      my $self = shift;

1516      # Open the LOGFILE for append >>.

1517      open ML_LOGFILE, ">>" . LOGFILE
1518          or die "Mobile::Location: unable to append to LOGFILE.\n";

1519      print ML_LOGFILE scalar localtime, ": @_\n";

1520      close ML_LOGFILE;
1521  }

1522  sub _logger2 {

1523      # This small routine quickly writes a message to the LOGFILE.  Note
1524      # that every line written to the LOGFILE is timestamped.  This code is
1525      # the same as "_logger", but for the fact that the location of the
1526      # LOGFILE is one-level-up in the directory hierarchy.
1527      #
1528      # IN:  a message to log.
1529      #
1530      # OUT: nothing.

1531      my $self = shift;

1532      # Open the LOGFILE (which is one-level-up) for append >>.

1533      open ML_LOGFILE, ">>../" . LOGFILE
1534          or die "Mobile::Location: unable to append to LOGFILE.\n";

1535      print ML_LOGFILE scalar localtime, ": @_\n";

1536      close ML_LOGFILE;
1537  }
```

```
1538  sub _build_index_dot_html {

1539      # Builds the INDEX.HTML file (used by _start_web_service).
1540      #
1541      # IN:  nothing.
1542      #
1543      # OUT: nothing (although "index.html" is created).

1544      my $self = shift;

1545      open HTMLFILE, ">index.html"
1546          or die "Mobile::Executive: index.html cannot be written to: $!.\n";

1547      print HTMLFILE<<end_html;

1548  <HTML>
1549  <HEAD>
1550  <TITLE>Welcome to the Location Web-Based Monitoring Service.</TITLE>
1551  </HEAD>
1552  <BODY>
1553  <h2>Welcome to the Location Web-Based Monitoring Service</h2>
1554  end_html

1555      print HTMLFILE "Location executing on: <b>" . hostname . "</b>.<p>";
1556      print HTMLFILE "Location date/time: <b>" . localtime() .
1557                          "</b>. Running on port: <b>" .
1558                              $self->{ Port } . "</b>.<p>";

1559      print HTMLFILE<<end_html;

1560  Click <a href="clearlog.html">here</a> to reset the log.
1561  <h2>Logging Details</h2>
1562  <pre>
1563  end_html

1564      open HTTP_LOGFILE, LOGFILE
1565          or die "Mobile::Location: the LOGFILE is missing - aborting.\n";

1566      while ( my $logline = <HTTP_LOGFILE> )
1567      {
1568          print HTMLFILE "$logline";
1569      }

1570      close HTTP_LOGFILE;

1571      print HTMLFILE<<end_html;

1572  </pre>
1573  end_html

1574      print HTMLFILE VISIT_SCOOBY;

1575      print HTMLFILE<<end_html;

1576  </BODY>
1577  </HTML>
1578  end_html

1579      close HTMLFILE;
1580  }
```

```
1581  sub _build_clearlog_dot_html {

1582      # Builds the CLEARLOG.HTML file (used by _start_web_service).
1583      #
1584      # IN:  the name of the just-created backup file.
1585      #
1586      # OUT: nothing (although "clearlog.html" is created).

1587      my $self = shift;

1588      my $backup_log = shift;

1589      open CLEARLOG_HTML, ">clearlog.html"
1590          or die "Mobile::Executive: clearlog.html cannot be written to:
                    $!.\n";

1591      print CLEARLOG_HTML<<end_html;

1592  <HTML>
1593  <HEAD>
1594  <TITLE>Location Logfile Reset.</TITLE>
1595  </HEAD>
1596  <BODY>
1597  <h2>Location Logfile Reset</h2>
1598  The previous logfile has been archived as: <b>$backup_log</b><p>
1599  Return to this Location's <a href="index.html">main page</a>.<p>
1600  end_html

1601      print CLEARLOG_HTML VISIT_SCOOBY;

1602      print CLEARLOG_HTML<<end_html;

1603  </BODY>
1604  <HTML>
1605  end_html

1606      close CLEARLOG_HTML;
1607  }

1608  sub _start_web_service {

1609      # Starts a small web server running on port HTTP_PORT.  Provides for
1610      # some simple monitoring of the Location.
1611      #
1612      # IN:  nothing.
1613      #
1614      # OUT: nothing.

1615      my $self = shift;

1616      my $httpd = HTTP::Daemon->new( LocalPort => HTTP_PORT,
1617                                     Reuse     => 1 )
1618          or die "Mobile::Location: could not create HTTP daemon on " .
1619                      HTTP_PORT . ".\n";

1620      $self->_logger( "Starting web service on port:", HTTP_PORT )
              if $self->{ Web };

1621      while ( my $http_client = $httpd->accept )
1622      {
1623          if ( my $service = $http_client->get_request )
1624          {
```

```
1625             my $request = $service->uri->path;

1626             if ( $service->method eq 'GET' )
1627             {
1628                 my $resource;
1629
1630                 if ( $request eq "/"  || $request eq "/index.html" )
1631                 {
1632                     $resource = HTML_DEFAULT_PAGE;

1633                     $self->_build_index_dot_html;

1634                     $http_client->send_file_response( $resource );
1635                 }
1636                 elsif ( $request eq "/clearlog.html" )
1637                 {
1638                     # Create a name for the backup log.

1639                     my $backup_log = "Mobile::Location." . localtime() .
1640                                          "." . $$ . ".log";

1641                     # Make the backup, delete the LOGFILE, then recreate it.

1642                     system( "cp", LOGFILE, $backup_log );
1643                     unlink LOGFILE;

1644                     $self->_logger( "Mobile::Location: log reset." )
                               if $self->{ Web };

1645                     $self->_build_clearlog_dot_html( $backup_log );

1646                     $http_client->send_file_response( "clearlog.html" );
1647                 }
1648                 else
1649                 {
1650                     $http_client->send_error( RC_NOT_FOUND );
1651                 }
1652             }
1653             else
1654             {
1655                 $http_client->send_error( RC_METHOD_NOT_ALLOWED );
1656             }
1657         }
1658     }
1659     continue
1660     {
1661         $http_client->close;
1662         undef( $http_client );
1663     }
1664 }

1665 sub _register_with_keyserver {

1666     # Create a PK+ and PK- for this server, storing the PK+ in the
1667     # keyserver, and retaining the PK- in memory (as part of the objects
1668     # state).  Note: a new key-pair is generated with each invocation.
1669     #
1670     # IN:  nothing.  (Other than the object reference, of course).
1671     #
1672     # OUT: nothing.
1673
1674     my $self = shift;
```

```
1675      # Generate the PK+ and PK-.  Store the PK- in the object's state.

1676      my $rsa = new Crypt::RSA;

1677      my $id = $self->{ Host } . ":" . $self->{ Port } . " Location";

1678      warn "This location is generating a PK+/PK- pairing.\n" if $self->{ Debug };

1679      my ( $public, $private ) =
1680              $rsa->keygen(
1681                  Identity  => $id,
1682                  Size      => KEY_SIZE,
1683                  Password  => $self->{ Password },
1684                  Verbosity => FALSE
1685              ) or die $rsa->errstr, "\n";

1686      warn "Pairing Generated.\n" if $self->{ Debug };

1687      $self->_logger( "Location's PK+/PK- pairing generated." )
              if $self->{ Web };

1688      # Remember the PK- in the object's state.

1689      $self->{ PrivateKey } = $private;

1690      # Write the PK+ to an appropriately named disk-file.

1691      my $pub_fn = $self->{ Host } . "." . $self->{ Port } . ".public";

1692      $self->_logger( "Writing PK+ to: $pub_fn." ) if $self->{ Web };

1693      $public->write( Filename => $pub_fn );

1694      # Determine the KEYSERVER address from the .scoobyrc file.

1695      open KEYFILE, SCOOBY_CONFIG_FILE
1696                  or die "Mobile::Location: unable to access ~/.scoobyrc.
                            Does it exist?\n";

1697      my $keyline = <KEYFILE>;

1698      close KEYFILE;

1699      # Note: format of 'rc' file is very strict.  No spaces!
1700      $keyline =~ /^KEYSERVER=(.+)/;

1701      $self->{ KeyServer } = $1;

1702      # Now that we know the address of the keyserver, we can register the
              PK+ of this
1703      # Location with the keyserver.  We read the PK+ from the just-created
              disk-file.

1704      $self->_logger( "Determined keyserver address as:",
              $self->{ KeyServer } ) if $self->{ Web };

1705      open KEYFILE, "$pub_fn"
1706          or die "Mobile::Location: KEYFILE does not exist: $!.\n";

1707      my @entire_keyfile = <KEYFILE>;
```

```
1708      close KEYFILE;

1709      my $keysock_obj = IO::Socket::INET->new( PeerAddr
                                                       => $self->{ KeyServer },
1710                                               PeerPort
                                                       => REGISTRATION_PPORT,
1711                                               Proto     => 'tcp' );

1712      if ( !defined( $keysock_obj ) )
1713      {
1714          die "Mobile::Location: could not create socket object to key
                  server: $!.\n";
1715      }
1716      print $keysock_obj $self->{ Port }, "\n";
1717      print $keysock_obj @entire_keyfile;

1718      $keysock_obj->close;

1719      $self->_logger( "Location registered with keyserver." )
               if $self->{ Web };
1720  }

1721  sub start_concurrent {

1722      # Start a passive server/location that executes concurrently.  For
1723      # each relocation request, a child process is spawned to process it.
1724      #
1725      # IN:  nothing.
1726      #
1727      # OUT: nothing.
1728      #
1729      # This method is never returned from.  Remember: servers are PERMANENT.
1730
1731      my $self = shift;

1732      my $listening_socket = IO::Socket::INET->new( LocalPort
                                                            => $self->{ Port },
1733                                                    Listen    => SOMAXCONN,
1734                                                    Proto     => 'tcp',
1735                                                    Reuse     => TRUE );

1736      if ( !defined( $listening_socket ) )
1737      {
1738          die "Mobile::Location: unable to bind to listening socket: $!.\n";
1739      }

1740      $self->_logger( "Location (concurrent) starting on port:",
               $self->{ Port } ) if $self->{ Web };

1741      warn "Location starting up on port: " . $self->{ Port } . ".\n"
               if $self->{ Debug };
1742
1743      $self->_register_with_keyserver;
1744
1745      while ( TRUE )  # i.e., FOREVER, as servers are permanent.
1746      {
1747          next unless my $from_socket = $listening_socket->accept;

1748          next if my $child = fork;

1749          if ( $child == FALSE )
1750          {
```

```
1751             $self->_logger( "Servicing client from:",
1752                                 inet_ntoa( $from_socket->peeraddr ) )
                                        if $self->{ Web };

1753             $listening_socket->close;
1754             $self->_service_client( $from_socket );
1755             exit FALSE;
1756         }

1757         $from_socket->close;
1758     }
1759 }
1760
1761 sub start_sequential {

1762     # Start a passive server/location that executes sequentially.
1763     #
1764     # IN:  nothing.
1765     #
1766     # OUT: nothing.
1767     #
1768     # This method is never returned from.  Remember: servers are PERMANENT.

1769     my $self = shift;

1770     my $listening_socket = IO::Socket::INET->new( LocalPort
                                                          => $self->{ Port },
1771                                                   Listen    => SOMAXCONN,
1772                                                   Proto     => 'tcp',
1773                                                   Reuse     => TRUE );

1774     if ( !defined( $listening_socket ) )
1775     {
1776         die "Mobile::Location: unable to bind to listening socket: $!.\n";
1777     }

1778     $self->_logger( "Location (sequential) starting on port:",
              $self->{ Port } ) if $self->{ Web };

1779     warn "Location starting up on port: " . $self->{ Port } . ".\n"
              if $self->{ Debug };
1780
1781     $self->_register_with_keyserver;
1782
1783     # Servers are PERMANENT.

1784     while ( TRUE )
1785     {
1786         next unless my $from_socket = $listening_socket->accept;

1787         $self->_logger( "Servicing client from:",
1788                             inet_ntoa( $from_socket->peeraddr ) )
                                    if $self->{ Web };

1789         $self->_service_client( $from_socket );
1790     }
1791 }

1792 sub _service_client {

1793     # Service the receipt (and re-execution) of a mobile agent on
1794     # this Location.
```

```
1795      #
1796      # IN:  A socket object to communicate with/on.
1797      #
1798      # OUT: nothing.

1799      my $self = shift;

1800      my $socket_object = shift;

1801      my $tmp_fn = <$socket_object>;  # The received filename.
1802      chomp( $tmp_fn );

1803      # We just want the name-part, so a little regex magic gives it to us.

1804      $tmp_fn = ( split /\//, $tmp_fn )[-1];

1805      my $tmp_linenum = <$socket_object>;  # The received line number.
1806      chomp( $tmp_linenum );

1807      my $data = '';

1808      # Receive the signature and mobile agent code.

1809      while ( my $chunk = <$socket_object> )
1810      {
1811          $data = $data . $chunk;
1812      }

1813      # We need to split out the signature from the $data so that we can
               verify it.

1814      ( my $agent_signature, $data ) = split /\n--end-sig--\n/, $data;

1815      # We need to verify the signature.  To do this, we need to retrieve
1816      # the appropriate PK+ from the keyserver.

1817      my $key_srv_sock = IO::Socket::INET->new(
1818                                               PeerAddr
                                                       => $self->{ KeyServer },
1819                                               PeerPort => RESPONDER_PPORT,
1820                                               Proto    => 'tcp'
1821                                             );

1822      if ( !defined( $key_srv_sock ) )
1823      {
1824          $self->_logger( "Unable to create a verify socket." )
                 if $self->{ Web };
1825
1826          die "Mobile::Location: unable to create a verify socket to
                 keyserver: $!.\n";
1827      }

1828      my $agent_ip = $socket_object->peerhost;
1829      my $agent_port = $socket_object->peerport;

1830      print $key_srv_sock "$agent_ip\n";
1831      print $key_srv_sock $agent_port;

1832      $key_srv_sock->shutdown( 1 );

1833      my $verify_data = '';
1834
```

```
1835      while ( my $verify_chunk = <$key_srv_sock> )
1836      {
1837          $verify_data = $verify_data . $verify_chunk;
1838      }
1839
1840      $key_srv_sock->close;
1841
1842      # This splits the signature and data on the SIGNATURE_DELIMITER
1843      # pattern as used by the keyserver.

1844      ( my $verify_signature, $verify_data ) = split /\n--end-sig--\n/,
                $verify_data;

1845      if ( $verify_signature eq "NOSIG" )
1846      {
1847          $self->_logger( "WARNING: The keyserver returned NOSIG." )
                    if $self->{ Web };
1848
1849          # We need to abort, as the keyserver does not have the requested
1850          # signature.  This is bad.

1851          $socket_object->close;

1852          exit 0;  # Short circuit.
1853      }

1854      open VERIFY_FILE, ">$agent_ip.$agent_port.public"
1855          or die "Mobile::Location: could not create verify key file: $!\n";

1856      print VERIFY_FILE $verify_data;

1857      close VERIFY_FILE;

1858      my $agent_pkplus = new Crypt::RSA::Key::Public(
1859                                  Filename => "$agent_ip.$agent_port.public"
1860                              );

1861      my $rsa = new Crypt::RSA;

1862      my $verify = $rsa->verify(
1863                                   Message    => $data,
1864                                   Signature  => $agent_signature,
1865                                   Key        => $agent_pkplus,
1866                                   Armour     => TRUE
1867                               );

1868      if ( !$verify )
1869      {
1870          $self->_logger( "WARNING: could not verify signature for:",
1871                               inet_ntoa( $socket_object->peeraddr ),
1872                                   "using $agent_ip/$agent_port." )
                                          if $self->{ Web };

1873          die "Mobile::Location: could not verify signature of received
                    mobile agent.  Aborting ... \n";
1874      }

1875      $self->_logger( "Signature verified for $agent_ip/$agent_port." )
                if $self->{ Web };

1876      # Remove the agents PK+ keyfile, as we no longer need it.
```

```
1877      unlink "$agent_ip.$agent_port.public";

1878      # At this stage, we have a mobile agent that is encrypted using the PK+
1879      # of this Location, and we have verified the signature to be correct.
1880      # We use this Location's PK- to decrypt it.

1881      my $plaintext = $rsa->decrypt(
1882                                Cyphertext => $data,
1883                                Key        => $self->{ PrivateKey },
1884                                Armour     => TRUE
1885                              );

1886      if ( !defined( $plaintext ) )
1887      {
1888          $self->_logger( "WARNING: unable to decrypt Cyphertext for:
                  $agent_ip/$agent_port." ) if $self->{ Web };

1889          die "Mobile::Location: decryption errors - aborting.\n";
1890      }

1891      # We have a plaintext representation of the mobile agent, which
1892      # we turn back into an array of lines.

1893      my @entire_thing = split /\n/, $plaintext;

1894      # Add a newline to each of the "lines" in @entire_thing.

1895      foreach my $line ( @entire_thing )
1896      {
1897          $line = $line . "\n";
1898      }

1899      # Ensure the Location is in the correct STARTUP directory.

1900      chdir $_PWD;

1901      # We enter the run-time directory if it exists.

1902      if ( -e RUN_LOCATION_DIR )
1903      {
1904          chdir( RUN_LOCATION_DIR );
1905      }
1906      else # Or, if it does NOT exist, we create it then change into it.
1907      {
1908          mkdir( RUN_LOCATION_DIR );
1909          chdir( RUN_LOCATION_DIR );
1910      }

1911      # As we are now in the run-time directory, we continue with the
              relocation.

1912      if ( $self->{ Log } )
1913      {
1914          my $logname = "last_agent_" . $$ . ".log";  # Note use of PID.
1915
1916          # Put a copy of the mobile agent into the log file.

1917          my $logOK = open AGENTLOGFILE, ">$logname"
1918              or warn "Mobile::Location: could not open log file: $!.\n";

1919          print AGENTLOGFILE @entire_thing if defined $logOK;
```

```
1920          close AGENTLOGFILE if defined $logOK;

1921          $self->_logger2( "Received agent logged to: $logname." )
                    if $self->{ Web };
1922      }

1923      # Untaint the filename received from Scooby, using a regex.

1924      $tmp_fn =~ /^([-\@\w_.]+)$/;
1925      $tmp_fn = $1;
1926
1927      # Create the "mutated" agent on the local storage.

1928      open FILETOCHECK, ">$tmp_fn"
1929          or die "Location::Mobile: could not create agent disk-file: $!:";
1930
1931      my $label = _generate_label( $tmp_fn, $tmp_linenum );
1932
1933      # Start processing the agent one "line" at a time.

1934      my $chunk = shift @entire_thing;

1935      # Print the "magic" first line.

1936      print FILETOCHECK $chunk;

1937  #    # Add the Opcode mask to the code.
1938  #
1939  #    print FILETOCHECK "\nuse ops qw( " .
1940  #
1941  #        # Basic operation mask - relocating to a single Location.
1942  #
1943  #        'aassign add aelem av2arylen ' .
1944  #        'backtick ' .
1945  #        'caller chdir chomp chop closedir concat const ' .
1946  #        'defined die ' .
1947  #        'enter entereval enteriter entersub eq ' .
1948  #        'ftdir fteexec ftewrite ' .
1949  #        'gelem goto grepstart gv ' .
1950  #        'helem ' .
1951  #        'iter ' .
1952  #        'join ' .
1953  #        'last leaveeval leaveloop leavesub lstat ' .
1954  #        'method method_named ' .
1955  #        'ne negate next not null ' .
1956  #        'open_dir ' .
1957  #        'padany pop push pushmark ' .
1958  #        'readdir refgen require return rv2av rv2cv rv2gv rv2hv rv2sv ' .
1959  #        'sassign scalar seq shift sne split stat stringify stub substr ' .
1960  #        'undef unshift unstack ' .
1961  #
1962  #        # Relocating to multiple Locations (requires more operations).
1963  #        # Most of these are needed by Carp.pm, which is used by IO::Socket
1964  #        # (among other modules).
1965  #
1966  #        'anonhash anonlist ' .
1967  #        'exists ' .
1968  #        'keys ' .
1969  #        'gt ' .
1970  #        'length lt ' .
1971  #        'mapstart ' .
1972  #        'ord ' .
```

```
1973  #         'postinc predec preinc ' .
1974  #         'redo ref ' .
1975  #         'sprintf subtract ' .
1976  #         'wantarray ' .
1977  #
1978  #         # Adding the ops required by Crypt::RSA and its support modules.
1979  #
1980  #         'anoncode ' .
1981  #         'bless bit_and bit_or bit_xor ' .
1982  #         'chr close complement ' .
1983  #         'divide delete dofile ' .
1984  #         'each enterwrite eof ' .
1985  #         'fcntl fileno flip flop formline fteread ftfile ftis ftsize ' .
1986  #         'ge getc ' .
1987  #         'hex '
1988  #         'int index ioctl ' .
1989  #         'lc le left_shift lslice '
1990  #         'modulo multiply '
1991  #         'oct open '
1992  #         'pack padsv postdec pow print prtf '
1993  #         'quotemeta ' .
1994  #         'rand read readline repeat reverse regcreset ' .
1995  #         'select splice srand sysread syswrite '
1996  #         'tell tie trans truncate '
1997  #         'uc unpack '
1998  #         'values vec '
1999  #         'warn '
2000  #         'xor '
2001  #
2002  #         $self->{ Ops } . " );\n\n";  # Forces safety.
2003  #

2004      # Insert the GOTO label line.

2005      print FILETOCHECK "goto $label;\n";

2006      # We re-initialize the line counter.

2007      my $line_counter = 2;

2008      # Process the rest of the agent, one "line" at a time.

2009      while ( $chunk = shift @entire_thing )
2010      {
2011          if ( $line_counter == $tmp_linenum )  # We are at the 'next' line.
2012          {
2013              # Insert a 'label' statement before the next instruction.

2014              print FILETOCHECK "$label:\n1;\n";

2015              print FILETOCHECK "use Mobile::Executive;\n\n";
2016          }
2017          print FILETOCHECK $chunk;
2018          $line_counter++;
2019      }
2020
2021      close FILETOCHECK;

2022      # Note: The agent now exists on the local run-time storage of this Location.
2023
2024      $self->_logger2( "Received $tmp_fn from", $socket_object->peerhost,
2025                       " next line: $tmp_linenum." ) if $self->{ Web };
```

```
2026      warn "Received $tmp_fn from ",
2027              $socket_object->peerhost,
2028                  "; next line: $tmp_linenum.\n" if $self->{ Debug };

2029      # Construct the command-line that will continue to execute the agent.

2030      my $cmd = "perl -d:Scooby " .  "$tmp_fn";
2031
2032      # Close the socket as we are now finished with it.

2033      close $socket_object
2034          or warn "Mobile::Location: close failed: $!.\n";

2035      # Continue to execute the agent at this location.

2036      warn "Continuing to execute agent: $cmd.\n" if $self->{ Debug };
2037
2038      $self->_logger2( "Continuing to execute mobile agent: $cmd." )
               if $self->{ Web };

2039      my $results = qx( $cmd );
2040
2041      print "$results" if $results ne '';
2042  }

2043  sub _spawn_web_monitoring_service {

2044      # Creates a subprocess to run the web-based monitoring service.
2045      #
2046      # IN:  nothing.
2047      #
2048      # OUT: nothing.

2049      my $self = shift;

2050      my $child_pid = fork;

2051      die "No spawned web-based monitoring service: $!.\n" unless
               defined( $child_pid );

2052      if ( $child_pid == FALSE )
2053      {
2054          # This is the CHILD code.

2055          $self->_start_web_service if $self->{ Web };

2056          exit 0;
2057      }
2058  }

2059  ###############################################################################
2060  # These are not methods, they're support subroutines.
2061  ###############################################################################

2062  sub _generate_label {

2063      # Generate a unique label string.
2064      #
2065      # IN:  A filename and a line number.
2066      #      Note: These values are combined with the time to produce a
2067      #      random (and hopefully unique) label.
```

```
2068      #
2069      # OUT: An appropriately formatted label.

2070      my $fn   = shift;
2071      my $ln   = shift;

2072      my $tm   = time;

2073      # Remove any unwanted characters from the filename.

2074      $fn =~ s/[^a-zA-Z0-9]//;

2075      return ( 'LABEL_' . $fn . $ln . $tm );
2076  }

2077  sub _check_for_modules {

2078      # Given a list of module classes, check to see if they exist within this
2079      # Location's Perl environment.
2080      #
2081      # IN:  A list of fully-qualified (one or more) module names.
2082      #      A "fully-qualified module name" is "Devel::Scooby", as
2083      #      opposed to just "Scooby".
2084      #
2085      # OUT: A list of modules NOT found.  An empty list signals SUCCESS.

2086      my @mods_to_check = @_;       # Taken from IN.

2087      my @list_of_not_found = ();   # Will be used as OUT.

2088      foreach my $mod ( @mods_to_check )
2089      {
2090          # Untaint the $mod values prior to their use, using a regex.

2091          $mod =~ /^([\w\d:_]+)$/;
2092          $mod = $1;

2093          eval "require $mod;";
2094          if ( $@ )
2095          {

2096              # The module does not exist within this Perl!!

2097              push @list_of_not_found, $mod;
2098          }
2099      }

2100      return @list_of_not_found;
2101  }

2102  sub _spawn_network_service {

2103      # Spawn a sub-process, running at protocol port number
              "$self->{ Port }+1"
2104      # to respond to an agent's query re: required classes.
2105      #
2106      # IN:  The protocol port to start the service on.
2107      #
2108      # OUT: nothing.

2109      my $port = shift;
```

```
2110      # Untaint the value for $port, as it can be initialized from
2111      # the command-line, and is therefore TAINTED.

2112      $port =~ /^(\d+)$/;
2113      $port = $1;

2114      my $child_pid = fork;

2115      die "No spawned network service: $!.\n" unless defined( $child_pid );

2116      # This child code never ends, as servers are PERMANENT.

2117      if ( $child_pid == FALSE )
2118      {
2119          # This is the CHILD code, which creates a server on "Port+1" and
2120          # listens for requests from a remote mobile agent.

2121          my $trans_serv = getprotobyname( 'tcp' );
2122          my $local_addr = sockaddr_in( $port, INADDR_ANY );

2123          socket( TCP_SOCK, PF_INET, SOCK_STREAM, $trans_serv )
2124              or die "Mobile::Location: socket creation failed: $!.\n";
2125          setsockopt( TCP_SOCK, SOL_SOCKET, SO_REUSEADDR, 1 )
2126              or warn "Mobile::Location: could not set socket option: $!.\n";
2127          bind( TCP_SOCK, $local_addr )
2128              or die "Mobile::Location: bind to address failed: $!.\n";
2129          listen( TCP_SOCK, SOMAXCONN )
2130              or die "Mobile::Location: listen couldn't: $!.\n";

2131           my $from_who;

2132           while ( $from_who = accept( CHECK_MOD_SOCK, TCP_SOCK ) )
2133           {
2134               # Switch on AUTO-FLUSHING.

2135               my $previous = select CHECK_MOD_SOCK;
2136               $| = 1;
2137               select $previous;

2138               my $data = '';
2139
2140               # Get the list of modules from the other Location.

2141               while ( my $chunk = <CHECK_MOD_SOCK> )
2142               {
2143                   $data = $data . $chunk;
2144               }

2145               my @modules = split / /, $data;

2146               my @list = _check_for_modules( @modules );

2147               if ( @list )
2148               {
2149                   print CHECK_MOD_SOCK "NOK: @list";
2150               }
2151               else
2152               {
2153                   print CHECK_MOD_SOCK "OK";
2154               }

2155               close CHECK_MOD_SOCK
```

```
2156                    or warn "Mobile::Location: close failed: $!.\n";
2157            }

2158            close TCP_SOCK;  # This code may never be reached.  It only
2159                             # executes if the call to "accept" fails.
2160        }

2161        # This is the parent process code.  That is, the value of
2162        # $child_pid is defined and is greater than 0.
2163    }

2164    1;  # As it is required by Perl.

2165    ##############################################################################
2166    # Documentation starts here.
2167    ##############################################################################

2168    =pod

2169    =head1 NAME

2170    "Mobile::Location" - a class that provides for the creation of Scooby
        mobile agent environments (aka Location, Site or Place).

2171    =head1 VERSION

2172    4.0x (the v1.0x, v2.0x and v3.0x series were never released).

2173    =head1 SYNOPSIS

2174    use Mobile::Location;

2175    my $location = Mobile::Location->new;

2176    $location->start_sequential;

2177    or

2178    $location->start_concurrent;


2179    =head1 SOME IMPORTANT NOTES FOR LOCATION WRITERS

2180    1. Never, ever run a Location as 'root'.  If you do, this module will die.
        Running as 'root' is a serious security risk, as a mobile agent is foreign
        code that you are trusting to execute in a non-threatening way on your
        computer.  (Can you spell the word 'v', 'i', 'r', 'u', 's'?!?)

2181    2. The B<Mobile::Location> class executes mobile agents within a restricted
        environment.  See the B<Ops> argument to the B<new> method, below, for more
        details.

2182    3. Never, ever run a Location on the same machine that is acting as your
        keyserver (it's a really bad idea, so don't even think about it).

2183    =head1 DESCRIPTION

2184    Part of the Scooby mobile agent machinery, the B<Mobile::Location> class
        provides a convenient abstraction of a mobile agent environment.  Typical
        usage is as shown in the B<SYNOPSIS> section above.  This class allows for
        the creation of a passive, TCP-based mobile agent Location.
```

```
2185  =head1 Overview

2186  Simply create an object of type B<Mobile::Location> with the B<new> method.
      To start a sequential server, use the B<start_sequential> method.  To start
      a concurrent server, use the B<start_concurrent> method.

2187  =head1 Construction and initialization

2188  Create a new instance of the B<Mobile::Location> object by calling the
      B<new> method:

2189  =over 4

2190  my $location = Mobile::Location->new;

2191  =back

2192  Optional named parameters (with default values) are:

2193  =over 4

2194  B<Debug (0)> - set to 1 to receive STDERR status messages from the object.

2195  B<Port (2001)> - sets the protocol port number to accept connections on.

2196  B<Log (0)> - set to 1 to instruct the Location to log the received mobile
      agent to disk prior to performing any mutation.  The name of the logged
      agent is "last_agent_PID.log", where PID is the process identifier of the
      Location.  On sequential Locations, the PID is always the same value for
      each received agent.  On concurrent Locations, the PID is the PID of the
      child process that services the relocation/re-execution, so it is always
      different for each received agent (so watch your disk space).  It is often
      useful to switch this option on (by setting Log to 1) when debugging.  Note
      that the received mobile agent persists on the Location's local disk storage.

2197  B<Ops ('')> - add a list of Opcodes to the Opcode mask that is in effect
      when the mobile agent executes.  Study the standard B<Opcode> and B<Ops>
      modules for details on Opcodes and how they are set.  One way to secure
      your Location against attack is to ensure that the Opcodes in effect while
      a mobile agent executes are "safe".  This is NOT an easy task, as protecting
      the mobile agent environment from malicious mobile agents is never easy.
      Note that the default set of Opcodes in effect are enough to allow the
      relocation mechanism to execute.  B<NOTE>: if the mobile agent uses a
      operation not allowed by the Opcode mask, it is killed and stops executing.
      The Location continues to execute, and waits passively for the next mobile
      agent to arrive.  The default set of enabled Opcodes is restrictive.
      Provide a space-delimited list of Opcodes to this argument to add to the
      list of allowed opcodes.  NOTE: this functionality is currently B<disabled>
      due to conflicts/incompatibilities with the current version of Crypt::RSA
      (version 1.50).

2198  B<Web (1)> - turns on the HTTP-based Monitoring Service running on port 8080
      (HTTP_PORT), thus enabling remote monitoring of the Locations current
      status.  It also logs interactions with this Location into 'location.log'
      (LOGFILE).  Set to 0 to disable this behaviour.

2199  =back

2200  Note that any received mobile agent executes in a directory called
      "Location", which will be created (if needs be) in the directory that houses
      this Location.  Any "logs" are also created in the "Location" directory.
```

```
2201  A constructor example is:

2202  =over 4

2203  my $place = Mobile::Location->new( Port => 5555, Debug => 1 );

2204  =back

2205  creates an object that will display all STDERR status messages, and use
      protocol port number 5555 for connections.  Logging of received agents to
      disk is off.  The standard Opcode mask is in effect.  And logging to disk
      is on, as is the HTTP server.

2206  When the Location is constructed with B<new>, a second network service is
      created, running at protocol port number B<Port+1>.  In the example above,
      this second network service would run at protocol port number 5556.  When
      sent the names of a set of Perl classes (e.g., Data::Dumper, HTTP::Request,
      Net::SNMP and the like), this service checks to see if the classes are
      available to the locally installed Perl installation.  This allows
      B<Devel::Scooby> to determine whether or not relocation is worthwhile prior
      to an attempted relocation.  The B<Devel::Scooby> module tries to
      determines the list of classes used by any mobile agent and communicates
      with this second network service "in the background".  This all happens
      automatically, so the mobile agent programmer does not need to worry about
      it, as B<Devel::Scooby> only complains when a module does not exist on a
      remote Location.  That said, the administrator of the Location does need
      to be aware of this second network service.  To confirm that the Location
      and the second network service are up-and-running use the B<netstat -an>
      command-line utility (on Linux).  The two "listening" services should appear
      in netstat's output.

2207  Note: If a Location crashes (or is killed), the second network service can
      sometimes keeps running.  After all, it is a separate process (albeit a
      child of the original).  Trying to restart the Location results in an
      "bind to address failed" error message.  Use the B<ps -aux> command to
      identify the Perl interpreter that is executing and kill it with
      B<kill -9 pid>, where B<pid> is the process ID of the child process's Perl
      interpreter.

2208  =head1 Class and object methods

2209  =over 4

2210  =item B<start_concurrent>

2211  Start the location as a passive server, which operates concurrently.  Once
      connected to a client, the server forks another process to receive and
      continue executing a mobile agent.  This is the preferred method to use
      when there exists the potential to have an agent execute for a long period
      of time.

2212  =item B<start_sequential>

2213  Start the location as a passive server, which operates sequentially.  Once
      connected to a client, the server sequentially processes the receipt and
      continued executing of a mobile agent.  This is OK if the agent is quick
      and not processor intensive.  If the agent has the potential to execute
      for a long period of time, use the B<start_concurrent> method instead.
      This may also be of use within environments that place a restriction on the
      use of B<fork>.

2214  =back
```

```
2215  =head1 Internal methods/subroutines

2216  The following list of subroutines are used within the class to provide
      support services to the class methods.  These subroutines should not be
      invoked through the object (and in some cases, cannot be invoked through
      the object).

2217  =over 4

2218  =item B<_generate_label>

2219  Takes a filename and line number, then combines them with the current time
      to produce a random, unique label.

2220  =item B<_check_for_modules>

2221  Given a list of module names, checks to see if the Location's Perl system
      has the module installed or not.

2222  =item B<_spawn_network_service>

2223  Used by the B<new> constructor to spawn the Port+1 network service which
      listens for a list of modules names from a mobile agent, then checks for
      their existence within the locally installed Perl system.

2224  =item B<_service_client>

2225  Given a socket object (and the instances init data), service the relocation
      of a Scooby mobile agent.

2226  =item B<_register_with_keyserver>

2227  Creates a PK+ and PK- value for the server, storing the PK+ in the keyserver,
      and the PK- in the object's state.

2228  =item B<_logger> and B<_logger2>

2229  Logs a message to the LOGFILE.

2230  =item B<_build_index_dot_html>

2231  Builds the INDEX.HTML page for use by the HTTP-based Monitoring Service.

2232  =item B<_build_clearlog_dot_html>

2233  Builds the CLEARLOG.HTML page for use by the HTTP-based Monitoring Service.

2234  =item B<_start_web_service>

2235  Starts a small web server running at port 8080 (HTTP_PORT), and uses the
      two "_build_*" routines just described.

2236  =item B<_spawn_web_monitoring_service>

2237  Creates a subprocess and starts the web server.

2238  =back

2239  =head1 SEE ALSO

2240  The B<Mobile::Executive> module (for creating mobile agents), as well as
```

```
      B<Devel::Scooby> (for running mobile agents).

2241  The Scooby Website: B<http://glasnost.itcarlow.ie/~scooby/>.

2242  =head1 AUTHOR

2243  Paul Barry, Institute of Technology, Carlow in Ireland,
      B<paul.barry@itcarlow.ie>, B<http://glasnost.itcarlow.ie/~barrypi/>.

2244  =head1 COPYRIGHT

2245  Copyright (c) 2003, Paul Barry.  All Rights Reserved.

2246  This module is free software.  It may be used, redistributed and/or
      modified under the same terms as Perl itself.
```