Agent-based Traffic Generation

Agent programming is a paradigm for distributed computing. A mobile agent is nothing more than a computer program that can move taking its state with it. Distributed tasks that occur in some order and depend on the outcome of each other are easily implemented with a single function.

In the article I will introduce the mobile agent programming paradigm. I will also show you how to reproduce scenarios and generate realistic and adaptable network traffic. These two problems map well to the mobile agent paradigm.

Middleware

Legitimate mobile agents require middleware to run on each host. Middleware is software that receives and executes an agent. Code that moves host to host with no middleware is called a worm. While these are equally fun, I’m not writing about them today.

Here I use examples written in the Sleep programming language. Sleep is an interpreter written on top of the Java virtual machine. I have two motivations for using it here. First, it is portable. Second, it supports a concept known as strong mobility. Strong mobility means a program can package its data, program counter, call stack, code, and transfer it elsewhere.

Most mainstream programming languages including Java are limited to weak mobility. Agent systems that rely on weak mobility cannot move the program counter and call stack. This places an unnecessary burden on the programmer to track state themselves. Unnecessary burdens translate to repetitive and cumbersome code. The lack of strong mobility support in mainstream languages has stunted the adoption and consideration of this useful technique.

Listing 1. Simple middleware

```
global('Server Agent');

while (!) {
    $server = listen(8888, 0);
    $agent = readObject($server);
    fork($agent, $agent);
}
```

Listing 2. Simple middleware cont.

```
inline move :
    call lambda();
local('$handle');
$handle = connect($host, 8888);
writeObject($handle, $1);
close($handle);
$host = $1;
```

agent middleware is three lines of code enclosed within a while loop. The listen function accepts connections on port 8888. Any waiting connections are queued by default.

After a connection is established, an agent is read with the readObject function. This function reads in a stream of bytes and reconstitutes an object from them. This process is known as deserialization. Converting an object to bytes is known as serialization.

The fork function creates a new thread and executes the agent object. The first parameter to fork is an anonymous function. Code wrapped in curly braces represents an anonymous Sleep...
function. The second parameter to fork is a value to pass into the global scope of
the new thread. Sleep isolates threads by default. After all, there is no need to protect
local data that isn’t shared.

You may be thinking, ‘what is an agent
and how do they work?’ The core object
is a paused sleep function. A function
requests to move itself by calling the move
function. Listing 2 shows the code for this
function.

Use the callcc command to
pause a function. You can read callcc
as call this anonymous function with a
continuation of the current function as a
parameter. A continuation is a paused
function. A paused function resumes
execution on the next call. This ability to
pause a function is key to the strong
mobility equation. Sleep functions
paused or not, are serializable. The
Sleep interpreter organizes the code, call
stack, variables, and program counter of
a function in one object. When a script
serializes a function it serializes this
whole package. This is how we achieve
strong mobility.

In the move function, an anonymous
function paused to callcc opens a
connection to a host on port 8888.
The writeObject function serializes the
continuation to the socket. I hide the
complexity of callcc behind the inline
function move. Inline functions execute
inline and a callcc within them affects
the caller. Agents specify the target host as a
parameter to the move function. Nothing
bother trying this out yourself. Place listing
1 and 2 into a file called middleware.s1. Then
type:

```bash
java -jar sleep.jar middleware.s1
```

Next, create an agent.s1 file that begins
with code from listings 2 and 3. Listing
3 shows a simple information gathering
agent. This agent collects information
about a host by executing the uname
command. Presumably it starts on
192.168.1.101. It prints this information
and moves to 192.168.1.102. It then gets
more information and saves it to $info. This
agent moves back to 192.168.1.101 and
prints the information from 192.168.1.102.
Add to agent.s1 this code. It launches the
agent is:

```
Listing 3. Uname agent

sub UnnameAgent {
  local($_info);
  $info = "uname"[0];
  printn("192.168.1.101 is $info");

  move("192.168.1.102");
  $info = "uname"[0];
  move("192.168.1.101");
  printn("192.168.1.102 is $info");
}
```

```
Listing 4. Middleware.s1

debuge[1 | 34];

include("lib/agentlib.s1");
include("lib/irclib.s1");

restoreAgents();

global("$server $agent");

while (1) {
  $server = listen(8888, 0);
  $agent = readObject($server);

  if ($agent["$name"] != null) {
    saveAgent($agent);

  }

  runAgent($agent);
}
```

```
Listing 5. Agentlib.s1 – checkpointing code

sub saveAgent {
  local("$handle");
  $handle = open("$handle", getFieldName("agents", $1["$name"]));
  writeObject($handle, $1);
  close($handle);

  inline save {
    callcc { saveAgent($1); };
  }

  sub runAgent {
    fork {
      ($agent);
      deleteFile(getFieldName("agents", $agent["$name"]))
      $agent => $1;
    }
  }

  sub restoreAgents {
    local("$handle $pump $name");

    foreach Name in("agents") {
      $handle = open(getFieldName("agents", $1["$name"]));

      while ($pump) {
        readObject($handle) {
          if (!$pump) ($agent = $pump);
          deleteFile(getFieldName("agents", $name));
          saveAgent($agent);
          runAgent($agent);
        }
      }
    }
  }
```
Once the code is in an agent, type:

```
java -jar sleep.jar agent.ai
```

This will launch the agent and you will see the output in the middleware window for 192.168.1.101. Figure 1 shows this.

So there you have it. These snippets contain the basic code necessary to implement agent middleware. I’ve hosted 1000 agents in this middleware on a normal Windows PC. The size of the agents depends on how much data they are carrying and the size of the code. The UnarmedAgent is 2KB. Listings 4, 5, and 6 contain the complete source code to the middleware used in the rest of this article. The next section explains additional features in this updated middleware to provide dependability in a test environment.

**Dependability**

If you’re planning to design agents that will run for long periods of time then

```
local(`$process`);
$agent = connect(`192.168.1.101`, 8888);
writeObject($agent, 4AgentAgent);
close($agent);
```

**Listing 6. Agentlib.ai – movement code**

```
sub sendAgent {
local(`$handle`, `$exception`);

while (1) {
  try {
    $handle = connect($1, 8888);
    writeObject($handle, 52);
    close($handle);

    if (-exists getFileInfo(`agents`, 52, `Name`)) {
      deleteFile(getFileInfo(`agents`, 52, `Name`));
    }

    return;
  } catch $exception {
    warn(`Error to $1 : $exception`);
    sleep($1 * 1000);
  }
}
```

```
inline move {
  alloc lambda{
    sendAgent($host, $1);
  }, $host => $1;
}
```

**Listing 7. Phishing scenario agent**

```
sub phish {
  local(`$victim`, `$exception`, `data` `$handle`);

  # move to victim box and connect it to IRC
  move(`192.168.1.200`);
  $victim = rand_word();
  connect_irc($victim, `192.168.1.200`);
  sleep(5000);
  # move to attacker box and connect attacker to IRC
  move(`192.168.1.10`);
  $handle = connect_irc(rand_word(), `192.168.1.107`);
  sleep(5000);
  # open the victim
  println($handle, `PRIVMSG $victim :Names! download this software! http://192.168.1.107/backdoor`);
  # move to victim computer
  $handle = $victim;
  move(`192.168.1.200`);
  # do a web request
  try {
    $handle = connect(`192.168.1.10`, 80);
    println($handle, `GET /backdoor`);
    readln($handle, `-1`);
    close($handle);
    # for giggles, pretend we were compromised,
    # start scanning network with snmp
    `snmp -V v2c 192.168.1.0.24`;
    println(`[SUCCESS] success`);
  } catch $exception {
    println(`[FAIL] Phishing Attack: $exception`);
  }
}
```

```
sendAgent(`192.168.1.200`, lambda {phish, $name => `phishing.com`});
```
dependability features become important. Without built-in recovery, the crash of one system will force you to bring everything down and relaunch all your agents losing any progress. This is not a fun situation. Fortunately adding features to prevent this isn’t too hard.

You can use checkpointing to deal with host failures. Checkpointing consists of saving agents to a file. The code for this is similar to the move function. In this implementation, agents are saved after migration and deleted following completion. Agents also have the option to call save to protect intermediate progress. Upon startup, the middleware’s first action is to restore all agents saved in files. Listing 5 shows the checkpointing functionality in the agent.c file.

Of course, a host failure creates problems for agents trying to communicate with it. The move function loops infinitely until the agent is successfully sent. This is crude but works fine in a lab environment. The improved movement code is in Listing 6.

These two techniques will let you recover from many failures by restarting the middleware on the problem host.

Scenario Coordination

Now that the middleware is out of the way let’s talk about applications. Common in the network security research field is demonstrating a capability or tool against a scenario. Conducting these demonstrations usually requires coordinating multiple hosts. One approach to this problem is to write programs for each host and use the command line to start the program on each machine. This is a poor man’s distributed system where you act as coordinator. Agents make coordinating a sequence of activity across multiple hosts trivial.

Here I use a mobile agent to simulate a successful phishing attack. Figure 2 contains a flow chart depicting the phishing attack. This attack involves an attacker and a victim. Both are connected to an Internet relay chat server (IRC). The attacker messages the victim. The victim then downloads something from the attacker’s URL and executes it. The actual download step may succeed or fail. The flow of this scenario is simplified for the sake of brevity.

The code in Listing 7 contains the agent implementation of the phishing attack. The agent contains the code to handle the role of the attacker and the victim in this scenario. The structure of the agent closely follows the phishing attack flow chart. The mobility of the agent enables this. Once the victim connects to IRC, the agent moves to assume the role of the attacker. Once the attacker is connected, the agent sends a message to the victim. Once the message is sent, the agent moves and becomes the victim again. The code in Listing 7 depends on the IRC helper code in Listing 8.

Notice that the victim nickname is randomly generated and saved. This information travels with the agent. With agents you can script scenarios that are as random or fixed as you like. Randomly generated values can travel with the agent for use in future parts of the process.

This phishing scenario shows how to encapsulate a flow chart into an agent. Imagine having agents that conduct business as usual. With a little disciplined programming these agents can validate the success or failure of each action taken. If an action fails, the agent can generate a message stating what failed and why. By assigning numbers to each type of failure and success you can use agents to provide metrics about how well a network configuration supports one or more processes.

Traffic Generation: Overview

A traffic generator is software that puts lots of packets on the wire. The purpose of a traffic generator is to create the noise and scale of a real network with no users and sometimes using a limited amount of hardware. One approach to this problem is to replay captured traffic. This is a valuable tactic for putting many realistic sessions on the wire each with their own state. There is also the advantage of scale. With a limited amount of hardware you can replay massive amounts of traffic. Unfortunately, replayed traffic is static. It can’t adapt to and report on changes in the test network.
The other approach for traffic generation are traffic simulators. These tools simulate the activity of users on real (or virtual) hardware and from this the activity the network traffic is created. This technique offers the most realistic possible traffic but scalability and complexity is an issue.

Mobile agents make possible a better traffic simulator. You can encapsulate arbitrarily complex scenarios into a single agent. Scale is achieved by creating multiple instances of the same agent with different parameters. Very little code offers convincing, adaptable, and measurable network traffic generation.

Simulating Multiple Hosts
Traffic generation is no fun if all agents have the same IP address. Requiring a virtual machine or hardware for each simulated host greatly limits scalability. Fortunately, in Linux it is easy to create virtual network interfaces to bind additional IP addresses.

On Linux you can bind a new address with:

```
$ /sbin/ifconfig device:x address
$ /sbin/_route add -host address dev
```

Here `device` is the network device i.e. eth0. The variable `x` represents a virtual device number. Each address should correspond to its own virtual device number. Begin with 0 and work your way up from there. And of course address is the address you want to bind. Note that these changes go away after rebooting so it helps to put these into a script.

Listing 9 demonstrates such a script. This script binds 127 addresses to a network interface. It even creates an empty file for each address in the /proc directory.
The `rand_ip` function in Listing 8 uses these empty files to indicate available addresses. I use this function in Listing 10 to make an IRC agent connect from a random address.

### On the ’Net

- [http://sleep.dashhine.org/](http://sleep.dashhine.org/) - Sleep download and documentation
- [http://sleep.dashhine.org/download/hakin9_agent.tgz](http://sleep.dashhine.org/download/hakin9_agent.tgz) - examples from this article

### Listing 9. Bind More IP Addresses

```
# java -jar sleep.jar randips.pl eth0 192.168.3

$device, $preExt = @ARGV;
mdir(“ips”);
for ($x = 1; $x < 128; $x++) {
    ($ip, $dev) = "$preExt $device:x $x”;
    touch ips/$ip
    /sbin/ifconfig $ip dev $ip;
    /sbin/ route add -host $ip dev;
}
```

### Listing 10. An IRC Agent

```
dbg(7 | 34);
include(“lib/agentlib.pl”);
include(“lib/irclib.pl”);
sub irc_agent | local(“$handle $ip $channel $messages”);
    $handle = connect_irc($rand_word(), rand_ip());
    sleep(1000);
    $channel = “#”.randa(1000) > 0.25, rand_word(), ’hottub’;
    print($handle, “JOIN $channel”);
    while ( !randa($channel is rand(1000)) < 999) {
        sleep(rand(30 * 1000));
    }
    $messages = "($PRIVMSG $channel :".rand_string(rand_word()),
    "PRIVMSG :".rand_word()." :".rand_string(rand_word())
    if (rand(1000)) > 999) {
        print($handle, “PART $channel”);
        break;
    } else {
        print($handle, rand($messages));
    }
    print($handle, “QUIT :” . rand_string(rand_word()));
    close($handle);
    sendAgent("192.168.1.200", lambda($this, $name));
    global(“$x”);
for ($x = 0; $x < 128; $x++) {
    sendAgent("192.168.1.1", lambda($irc_agent, $name => rand_word() . rand(100)));
```

![Figure 3. Simulated Hosts Communicate with IRC Server]
Sleep's connect and listen functions let you specify which address to bind to. Use the laddr named parameter to do this. For example connect (IPv4 or IPv6 laddr = 10.101.8) connects to 192.168.13 on port 6667 using 10.101.8 as the outgoing address. And listen (IPv6 or IPv4 laddr = (IPv6 or IPv4 laddr = 192.168.13) listens on port 6667 of the interface where 192.168.13 is bound. With these functions and virtual devices you can easily simulate actions amongst multiple hosts.

Listing 10 shows the code for an agent that connects to IRC. This agent connects to a server and joins a channel. It then chooses to send a private message, channel message, quit the server, or part the channel. When the agent completes an IRC session, it starts a new copy of itself. This assures the agent is always connected or in the process of connecting to IRC. Figure 4 shows an Ethereal screenshot with 100+ such IRC agents. To create this traffic required one computer to act as a server and another to host the clients. Not bad. This technique works with other protocols as well.

Fully simulating a network protocol with connect and listen is cumbersome. One of the advantages of a Java based scripting language *cough*Sleep*cough* is the availability of multiple libraries for different protocols. The Sleep homepage and blog contain examples for other protocols including HTTP and SSH.

Unfortunately few of these libraries offer the flexibility to select which local addresses to bind outgoing connections to. This is the case even with internal libraries such as Java.net.URLConnection. If you are a strong Java programmer it isn't much work to add this option when the source code for a package is available. However, I realize source hacking isn't an option for everybody.

Another option is to create multiple middleware processes and limit each to a specific local IP address. This is accomplished by isolating the process at the kernel level. The Linux VServer project provides the support needed for this on Linux. In this way you're using light-weight virtualization to simulate multiple hosts. It is still more light weight than multiple virtual machines. Also the mobility of the agents is an asset here as well.

The agent can migrate between middleware instances with the move function.

**Conclusion**

In this article, I've introduced you to programming with mobile agents. My inspiration to use mobile agents for traffic generation came from a need to score students during a network security game.

My first requirement was to score students on the confidentiality, integrity, and availability of services. The agents generated data and followed it throughout its life cycle interacting with the student services. For example, an e-commerce agent would generate a false order, place it at a student run website and later move to an inside computer to process this order. If the order was unable to go through (availability) at any time or changed in any way (integrity) the agent would note this. To measure confidentiality we gave students a place to provide stolen files. The agents would move to this location and look for their data (confidentiality).

The second requirement was to prevent student tampering. As you can see, this middleware has no security. My solution? We used hardened Linux servers within each possible enclave. Each team had a server and the outside had a server. Each server had two network interface cards. One for an out-of-band network were the agents migrated. The other was for the competition traffic. Each middleware listened for migration traffic on the out-of-band interface.

The last thing I sought was scale and realism. As shown in this article the agents interact with the services just as a human would. The idea that the agents can coordinate and simulate a process with multiple actors provides the realism. The ability to measure and report the breakdown of this process and why provides metrics.

With agents, you can simulate both legitimate and malicious activity. With these techniques, you can start to ask questions about your network and design proper experiments.

**Raphael Mudge**
Raphael is a code master based in the United States. You can find out more at http://www.hack.org/~raf/