Active Probes: A Framework for Monitoring and Debugging
Active Programs (Release 1.6)

Jeff Kann, Bob Braden, Bob Lindell
USC/Information Sciences Institute
isi-arp@isi.edu

February 18, 2003

Contents

1 Introduction ........................................ 2

2 The Probe Framework .................................. 3
   2.1 Outline of Probe Mechanism .................... 3
   2.2 Probe Algorithms ................................ 6
   2.3 Probe Packet Format ............................ 7
   2.4 Client GUI ...................................... 9

3 Probe Implementations for the ASP EE ............. 10
   3.1 Probe for Jsrvp/AFSP .......................... 11
   3.2 ASP EE Probe GUI ............................... 13

4 Open Issues ........................................ 14
1 Introduction

Using active networking technology, network-oriented programs can be loaded and executed on routers and hosts to dynamically change the behavior of these nodes. The ability to readily introduce new programs and to change existing programs dynamically, combined with the distributed nature of active network algorithms, inevitably makes debugging and monitoring difficult for active programs.

Debugging and monitoring will be needed both for the occasional development of a new active networks Execution Environment (EE) and for the continuing development of persistent Active Applications (AAs) that execute in the existing EEs. Debugging is imagined to happen only once, while monitoring is imagined to be a continuing activity to detect operational anomalies. In reality, debugging and monitoring are likely to be interspersed throughout the development, deployment, and operation of successive generations of a protocol implementation.

This note describes a framework for active network monitoring and debugging that is based on a client sending an active program fragment to a target EE or AA on a remote node, to monitor and report back arbitrary pieces of information about the target’s execution state. We call such an active code fragment an active probe program, or simply a probe. An active probe program can compute an arbitrary function of the state contained in the target EE or AA. This might be a simple filtering function, reporting only specific subsets of the state, e.g., the state that has changed within some interval. It might be a more complex function that tests for anomalous conditions, reporting only when such conditions are found. Like other active code, probes should be easily modifiable and deployable.

The usual approaches for monitoring and/or debugging a running program (EE or AA) can be classified into three categories.

1. Brute force: stop program execution, modify the source code, insert debugging statements, recompile and re-deploy the new code, and finally observe the logs during the execution.

   Logging some debug output to monitor the long-term correctness of execution is often useful, as it may provide the first clue before the developer tries anything else. However, the brute force approach is not adequate in general for debugging active network programs (AAs). It is not practical to repeatedly shut down a node and reload the running program just to add debugging output.

2. Polling - send a query to request information existing in the program and analyze the replies.

   Polling is basically the query/response approach of SNMP-based network management. However, polling only works for debugging if the user happens to query a node at the “right moment.” Even when one query does not reveal any problem, the target program may fail without explanation before the next query. In addition, polling does not catch node state changes; most of the time the query either works successfully or totally fails because the damage is already done.

---

1Since we chose the name “probe”, we became aware that BBN’s SENCOMM project uses the same term for a related active network management function. We are currently undecided how to resolve this conflict.
3. Spying: deploy probes on some or all of the target nodes to report anomalies or node state back to the client. The probe reports may be sent whenever state changes or periodically at configured intervals.

Although all three approaches have an important role, only the first two are widely used. This paper concentrates on the third approach, using active probe programs.

Monitoring and debugging are really special cases of the network management problem. Probes may be classified as a form of active network management, in two senses: the managed entity is an active EE or AA component, and the management mechanism itself uses active network technology. However, probes focus on the active components — EEs and AAs — not on the nodeOS or the network environment. Therefore, the active probes described here may be expected to complement other research activity in network management, which has been focused on the more traditional aspects of the node software.

Section 2 describes the general probe framework. Section 3 describes an initial prototype implementation of probes for the ASP EE [1]. Finally, Section 4 presents some open issues.

2 The Probe Framework

2.1 Outline of Probe Mechanism

A probe may be an arbitrary program, within the constraints imposed by the language and execution environment (EE). Probe code will generally be target-specific; a probe must generally be written specifically for the particular AA or EE within which it will execute and on which it will report. A probe is useful precisely because it executes within the target EE or AA software environment, so that it has access to all the state variables that the client might wish to monitor. Depending upon the language environment, the probe might have be read/write or read-only access to the AA state variables.

A probe might be transient, gathering specific data and reporting it, or perhaps setting state, and then exiting immediately. Alternative, it might be persistent, continuing until it is terminated by the client. The persistent case is more interesting and general. For example, a persistent probe can monitor an AA for anomalies over a long period, without constant polling from the client. This memo generally emphasizes persistent probes, under the assumption that a transient probe is a special case.

A probe generally incorporates both code and data. The data fields are parameters that control how the probe operates; we call this configuration data. To install or terminate a persistent probe and to set or modify its configuration data, the client sends a probe control message to the target EE/AA on the target node. A probe control message can also be used to install a temporary probe, which will perform its function and then terminate itself.
To initiate a probe, the client’s user application sends a probe control message containing an INSTALL command to the target AA or EE. The target must be able to distinguish the probe control message from other active packets it receives; the method of distinguishing is EE-dependent.

A probe control message is addressed to the target node, EE, and AA, just like any other active packet. The addressing information would include an ANEP TypeID or whatever other field is needed to demux it to the target EE once it reaches the target node, and some further EE-specific demultiplexing to an AA. If a probe for a particular AA arrives before that AA has been loaded, the probe needs to carry enough information to allow the EE to load that AA before delivering the probe to it.

Transport of an active probe message to a target EE/AA might be done in any of the following ways.

- **Point-point In-band**
  
  From the “edge” of the network, the client’s User Application (UA) can launch a probe control packet towards the target node, as it would send any other active packet. This packet will be forwarded to the target node by intermediate active nodes, and any responses can be forwarded back to the client in the same manner.

- **Hop-by-Hop In-band**
  
  With the preceding approach, the client must send a distinct control message to each target node. Some EEs may also support “hop-by-hop” forwarding, in which an active packet that is addressed to an end-point can be deiverted and processed in each active node along the path. This allows a single active probe command to be involed for each node along a path.

- **Point-point Out-of-band**
  
  An existing infrastructure of non-active legacy protocols (e.g., TCP/IP) can be used to send a probe directly to the target node, out-of-band from active packet forwarding. For example, the client program can act as a remote user application (remote UA), sending the probe message directly to the UA interface of the EE in the target node. The client might open a TCP connection to a well-known port on the target host and sends the probe control messages. This approach will be important for debugging when active network forwarding is dysfunctional.

If the last approach is used for transport, a TCP connection will provide a signal to the other end should the client process or of the remote EE process terminate. If UDP is used for probe control messages, on the other hand, probes must use soft state refreshes or time out.

The probe framework operates in the following manner.

- **Receipt of an INSTALL message causes the AA/EE to load and instantiate the probe code and initiate its execution, if the specified probe does not already exist at the target. For example, under Java the probe might be started under a new thread, or it might use an existing AA/EE**
thread. Loading an AA may be accomplished by a library routine, or perhaps it will be possible to push it down into the EE (see Section 4).

- A particular probe will be programmed to monitor the execution of the target AA/EE and report changes of state or specific events back to the client. This reporting might be done in a real time or a polling basis, for example.

- The client can change the configuration data of an installed probe by sending a new INSTALL message for the same probe with new configuration data. This will be much more convenient than stopping the old probe and starting a new probe with different arguments.

- If the probe uses soft state, the client must send it periodic INSTALL messages to refresh the state. These messages need not include configuration data, unless it is to be changed.

- When the probe is no longer necessary, the client can send a TERMINATE message, instructing the probe to exit its infinite loop and exit. Whether the probe code is actually removed from storage depends upon the details of the EE.

A probe control message will be a 4-tuple:

(probeSpec, probeID, Command, configData)

These components are as follows:

- probeSpec
  The probeSpec specifies how the EE can find and load the probe code if it is not already loaded. Its syntax and semantics are EE-specific. For a Java-based EE, the probeSpec is expected to include the fully qualified name of a Java class.

- probeID
  The probeID is used to distinguish different probes from the same client with the same probeSpec. Thus, an installed probe is uniquely defined by the client’s transport-level address (to which replies will be returned), the probeSpec, and the probeID.

- Command
  The Command code selects the operation to be performed. It may be one of:
  
  - INSTALL: install the probe with specified configData.
  - TERMINATE: terminate an existing probe.

  for a persistent probe, or simply INSTALL for a transient probe.

- configData
  Probe-specific configuration data variables. Normally omitted for TERMINATE command or a refresh INSTALL command.
Note that the probeSpec element of the probe header is specified as a reference that is used to find and load the probe code out of band. It would be possible for the probe framework to encompass EEs that support only capsules, i.e., active code carried directly in active packets. However, with only capsule-based probe messages, the maximum packet size may place a severe limitation on the size and complexity of the probe code. This limitation would reduce the flexibility and the range of useful services that a capsule probe could perform.

Within the probe framework, a probe reply message can have any format that the probe and the client agree upon. However, again some standards are worthwhile. A probe reply message should be a 3-tuple:

(probeSpec, probeID, replyData)

We furthermore suggest standardizing the common case of replyData consisting of a set of character strings, to allow a generic GUI to display probe replies that choose to follow this convention.

2.2 Probe Algorithms

The particular function of a probe will generally be highly dependent on the circumstances. However, one important kind of probe application is to monitor particular state variables and report a change or recompute some test predicate.

A persistent probe might simply report state periodically, with an inner loop of the form:

```java
while( true ) {
    sleep(time_period);
    for all <interesting state>
        send_report(<interesting state>);
}
```

Alternatively, a persistent probe might immediately report state changes. This could be accomplished by explicitly inserting monitoring code into the AA. For example, the following changes might be inserted into a Java-based AA for probing its state.

- Add a new Boolean flag (state_mod to each data structure (object) whose state changes are to be monitored.
- Change every relevant AA class so that when it creates, modifies, or deletes such a structure, it sets this Boolean flag true and then issues a synchronized signalAll().

The probe thread, when awakened, must test all the monitored data structures, process any with the bit set, and turn off the bit again. Thus, it might then execute an inner loop of the form:

```java
while ( true ) {
```
wait_for_signal();
while ( <some state has mod flag on> ) {
    send_report( <state with mod flag on> );
    <turn off mod flag>;
}

The AA changes just described, while individually small, may be numerous; a couple of statements may have to be be inserted at every point where the AA modifies the monitored state. Fortunately, there is may be a more economical approach to monitoring changes in data structures. Some EE’s include a state repository or tuple storage mechanism. It should be possible to extend this mechanism to provide a standard interface for monitoring state changes. The idea is that a probe would be able to register a routine to be upcalled when any, or a specific subset of, the state changes in the repository. The probe code could then generate a probe reply packet for the client.

2.3 Probe Packet Format

The following diagram shows a proposed standard format for a probe command message from a client to a target node. Note that the framework does not require such standardization; active networking allows the use of any probe-specific protocol between a client and its probes. However, uniformity and simplicity may result from having a standard format, which can be built into probe library routines.

2.3.1 Probe Control Message Format

```
+----------------------------------+
| Probe Cookie                    |
+----------------------------------+
| Version        Command           |
+----------------------------------+
| ProbeSpec length  ConfigData length |
+----------------------------------+
// ProbeSpec  ......................  //
// ConfigData  ......................  //
```

Probe Cookie

The first few bytes of many protocol headers carry data patterns that may be used to distinguish a probe from other active packets. The first four bytes of the probe packet is reserved for a Probe Cookie, some AA-dependent value that may identify the packet as a probe. The default value for this field is zero.
The Probe Cookie is not the only way that an EE or AA can distinguish probes; for example, a separate inChannel might be opened to receive probe command messages. However, the Probe Cookie field is available for those cases where there is no more elegant alternative.

Version
The version of the probe packet format. This is version 1.

Command
Integer encoding the probe command to be executed: 1 = INSTALL, 2 = TERMINATE

Probe ID:
A halfword value that identifies the probe among multiple probes from the same client and probeSpec.

ProbeSpec length:
The total length of the probeSpec in bytes.

ConfigData length:
The total length of the ConfigData in bytes. Zero if there is no ConfigData.

ProbeSpec:
The Java class name for the probe class. It is padded to the nearest fullword boundary with zero bytes. Otherwise, its format is EE-dependent.

ConfigData:
An argument string defining configuration data. Its format is given below. It is padded to the nearest fullword boundary with zero bytes.

The ConfigData field contains an argument that allows the probe’s configuration variables to be set initially or changed by an INSTALL command. The configuration data is contained in a set of public variables that are defined in the probe class.

We propose the following standard form for ConfigData field: an ASCII string containing a set of simple assignment statements of the form:

<config variable name>=<constant value>

separated by Telnet newline sequences (CR LF). Here <constant value> will use an encoding appropriate to the type of <config variable name>. The particular encoding of variable names and constant values is specific to the language environment of the EE.

2.3.2 Probe Reply Format

Version
The version taken from the probe control message that installed the probe that created this reply.

Reply Format

Although the replay data format can be strictly private between the probe and the client, it may be useful to standardize on a few common probe reply formats. This field is used to indicate a registered format.

- Reply Format = 0: No format specified
- Reply Format = 1: Reply Data is list of N zero-terminated character strings, where N is the Reply Count field.
- Reply Format > 1: Available for future assignment.

Probe ID:
Same as in control message.

ProbeSpec length:
Same as in control message.

Reply Count:
This value indicates the number of segments in the Reply Data. If there are not multiple segments, it can have the value 1 or 0.

ProbeSpec:
Same as in control message.

Reply Data
The data being returned by the probe.

2.4 Client GUI

A probe client GUI would support the installation, configuration, and removal of probes in any probe-capable AA in a remote active node. Such a GUI would allow the client to send an INSTALL
or `TERMINATE` control message to the target, specifying the target node, EE, and AA, the probeSpec, and ConfigData as needed, and display the contents of probe reply messages. This GUI could be independent of the specific probe, but it will include components that are specific to a particular EE. In particular, the encoding of EE and AA selectors and perhaps the probeSpec will generally be EE-dependent. It may be possible to develop a table-driven mechanism for generating these strings for any EE.

A prototype probe GUI for the ASP EE is described below.

3 Probe Implementations for the ASP EE

This section describes a prototype active probe packet mechanism that has been implemented experimentally in the ASP EE. Because it was a prototype, the ASP EE probe code differs somewhat from the specific packet formats given earlier.

Probe command messages are always sent to an ASP EE on the target node out-of-band. The client opens a TCP connection to a well known port on the target host and passes the probe messages. These messages are delivered to the target AA via an upcall to `receivePacket()` for the "api" inChannel. Using TCP for such out-of-band probe signaling allows a probe to detect when the client has terminated without sending an explicit `TERMINATE` command. The `receivePacket()` upcall passes a length -1 to indicate that the API channel’s TCP connection has closed. This allows the AA to terminate all probes for that client.

A probe command message will be encapsulated with an ASP header containing the `AA.spec` that defines the AA and how to load it. If an `INSTALL` command packet arrives at the EE before the target AA has been loaded, the ASP EE will use the `A.spec` to load the AA before passing the `INSTALL` command to it.

The ASP EE currently implements a third command, `SET_PARAMS` to change the configData of an existing probe. However, this is not logically necessary. It does not require soft state refreshes, and simply ignores `INSTALL` messages when the probe is already installed.

Two probe classes have been implemented. The RsvpProbe class is a probe for Jrsvp, the AA implementation of the QoS reservation protocol RSVP. The JripProbe class is a probe for Jrip, the AA implementation of the RIP routing protocol. Each probe class extends the abstract class apps.lib.AAProbe (to differentiate it from asp.lib.EEProbe in the future) and implements the interface java.lang.Runtime.

- JripProbe currently uses a zero Probe Cookie, since a RIP protocol packet always has a non-zero value in the first two bytes of the payload.
- RsvpProbe uses a reserved API UDP port (9876) for probe packets. This information is available to the AA in the attribute object set by the `receivePacket()` upcall.
3 PROBE IMPLEMENTATIONS FOR THE ASP EE

3.1 Probe for Jrsvp/AFSP

The RsvpProbe class defines an active probe program may be used to monitor the state in RSVP/AFSP. This probe will report when any PSB or RSB state block for the given session is added, modified, or deleted. If verbose is true, it will also return a formatted dump of each such state block.

The configuration data in RsvpProbe is composed of eight fields; the first three control the operation of the probe, while the last five select the RSVP session to be probed.

    // Timeout in milliseconds
    public long timeout = 0;

    // Whether an immediate report is requested.
    public boolean immediate = true;

    // Verbose means more output to the client UA.
    public boolean verbose = false;

    // Address family for the session to be monitored.
    public String address_family = "AF_INET";

    // Session destination address
    public String dst_addr = "224.23.23.23";

    // Session UDP port number
    public int port = 33333;

    // Session protocol number
    public int protocol = 17;

    // Session GPI indication
    public boolean isGPI = false;

The methods in RsvpProbe build three hash tables:

- probeInfCache maps the API channel into the probeID.
- probeCache maps a probeID into a probeSpec.
- threadCache maps a probeID into a probe thread.

The first is used to locate the probeID if the API TCP connection closes prematurely.

The RsvpBase.probeInstall() method calls AspSystem.forName to load the probe class, using the probeSpec field as the class name. It then creates and initializes an instance of this probe class and starts the probe object running on a new AspThread.
The probe maps the configuration variables into a session, and thenceforth operates only on that session. Changing the session configuration with a SET PARAM command will have no effect.

If the configuration variable immediate is true, the inner loop of the probe executes the following sequence:

```c
while(true) {
    Synchronized(session) {
        wait(timeout);
        while (report(session)) {
            saveCurrent(session);
        }
    }
}
```

That is, it waits for a signal indicating that a PSB or RSB has changed. The routine report() scans all the PSBs and RSBS of the session, making a list of all the changes. If it finds any changes, it uses the list to build and send a report message and returns true; otherwise, it returns false. In each cycle, saveCurrent() is called to save a complete copy of the current PSB and RSB state; this is used to detect when a PSB or RSB has been deleted.

If the configuration variable verb+immediate+ is false, the inner loop executes the following sequence:

```c
while(true) {
    Synchronized(session) {
        if (timeout <= 0) {
            timeout = 30*1000; // default timeout = 30 seconds
            AspThread.sleep(timeout);
            send_report(session);
            saveCurrent(session);
        }
    }
}
```

That is, it executes a simply timing loop, polling for changes at a rate determined by timeout, with a default of 30 seconds if timeout is not positive.

The following modifications were made in the Jrsvp code to support RsypProbe.

- A new Boolean flag (ps_mod or rsb_mod) was added to each PSB and RSB.
- Every Jrsvp class that can create, modify, or delete a PSB or RSB was modified to set the Boolean flag true, and to then issue a synchronized signalAll1O when the corresponding state changed.

The probe thread, once awakened, will call report(session), which will turn off the ps_mod or rsb_mod bits.
3.2 ASP EE Probe GUI

The probe GUI for the ASP EE operates by making a TCP connection directly to the UA API of the ASP EE on the remote node. To run this GUI, type this command:

   java ua.afsp.SpyGUI

A Java GUI will prompt for the DNS name of the target node, the Java class name of the probe class, and configurable parameters. The probe class must be in the directory of the AA, so the probe class name implicitly specifies the AA. For example, the client might enter:

   Install on node: eye.isi.edu
   Probe Class Name: apps.rsvp.RsvpProbe

The configurable parameters prompt consists of a series of variable/value pairs. These pairs are obtained from the probe class. They show all the public non-final variables (that are configurable, as the name "Configurable Parameters" suggests). Unfortunately, there is currently no way to add comments to describe each variable and its default value. The best way to understand what these variables mean and what the desired values are is to read the probe code.

Next, there are three buttons to select the command to be performed: Install, Set Parameters, and Quit/Terminate.

- **Install**
  
The Install button will install the probe on the specified target node. After install is clicked, this button will be grayed out so you won't install it again by mistake. It will be shown again if you click on "terminate."

  ((Note that RsvpProbe ignores duplicate INSTALLs))

- **Set Parameters**
  
The Set Parameters button will reset the probe variable/value pairs at run time. For example, you might decide not to get immediate notification, so you can change the "timeout" and "immediate" variables to "30000" and "false", respectively, to obtain a 30 second reporting interval. Before the probe is installed, this button is grayed out.

- **Quit/Terminate**
  
The Quit/Terminate button will change its text to indicate its function. When the probe is *NOT* installed, this button shows "Quit"; clicking on it will quit the GUI. After you have installed the probe to a node, this button will show "Terminate", allowing you to uninstall the probe (i.e., stop its probe thread); the button will change itself back to "Quit."

Finally, there is a text area in which reply output is displayed.
[Unfortunately the "TextArea" component does not scroll itself to the bottom of the output, so the user has to use the scroll bar for that purpose in order to see the latest information.

By default, the "verbose" variable in the probe is set to "false" and the probe will only report simple status (for example, "PSB Added" or "RSB modified"). If you change "verbose" to be "true", it will also show the PSB and RSB details.

The prototype currently uses the following conventions for configData strings. The assignment statements are separated via the character '&amp;' and each space is replaced by a character '+'. For example, the following option string might be sent:

\[ \text{dst_addr=local&amp;comments=I+am+a+spy&amp;timeout=30&amp;immediate=false} \]

for a probe class that included the following configData variables:

\[
\begin{align*}
\text{public long timeout = 0;} \\
\text{public boolean immediate = true;} \\
\text{public String comment = "";} \\
\text{public String dst_addr = "224.23.23.23";}
\end{align*}
\]

4 Open Issues

1. Generalized Integration into ASP EE
   It should be possible to integrate probes more fully into the ASP EE in a very general fashion by adding an additional level of naming AAs. The ASP EE might receive active packets carrying AAspecs with an AAname of the form: "aaaaaa/subName". Here "aaaaaa" is the AAname for the AA alone, and "subName" names some subprogram to be loaded and executed within that AA. The base class in the AAspec would be the initial class for the subprogram. The AA itself would be loaded first if necessary. In the case where the subprogram was a probe, "subName" would name the probe code and the base class would be the probe class. The original AA would coexist with any other subprograms (probes) as just another distinctly named subprogram.
   [The mechanisms we use for probes seem to closely parallel those of SENCOM, although the objectives are fairly different.]

2. Capsule-Based Probes
   It may be worthwhile to further explore the space of capsule-based probes as an alternative to out-of-band loading. For example, Bob Lindell wrote:

   I have integrated the bean shell (www.beanshell.org) into a version of ASP. This provides the ability to execute interpreted Java code. One use, and my motivation, is command line debugging. But it could be used for probes. Probes could be
compressed Java source code programs. If you strip out comments, white space, and then compress the source, it should be possible to put a reasonable sized program into a single packet. It is clearly a win over compressed byte code.

References