

# Active Network Monitoring and Control: The SENCOMM Architecture and Implementation

Alden W. Jackson, James P.G. Sterbenz,  
Matthew N. Condell, Regina Rosales Hain  
[awjacks, jpgs, mcondell, rrhain]@bbn.com  
<http://www.ir.bbn.com/projects/sencomm/>

BBN Technologies  
10 Moulton St, Cambridge, MA 02138-1191\*

## Abstract

*We present the architecture, design, and implementation of a Smart Environment for Network Control, Monitoring and Management (SENCOMM). SENCOMM uses active network technology to comprise a Management Execution Environment (SMEE), which coexists with other execution environments (EEs). Management applications, called smart probes, run in the SMEE. A probe and its data are mobile executable code that are delivered to the active node within an Active Network Encapsulation Protocol (ANEP) datagram.*

*Our architecture is designed to actively control, monitor, and manage both conventional and active networks, and be incrementally deployed in existing networks. We present a set of goals, a design philosophy, and a set of basic requirements for controlling, monitoring, and managing networks using active network technology. We discuss the operation and components of SENCOMM: the management EE, a protocol, smart probes, and loadable libraries. We discuss the implementation issues uncovered in integrating SENCOMM into a selected EE and the decisions made to resolve them.*

## 1 Introduction and Motivation

### 1.1 Introduction to Active Networks

Active Networking (AN) is an emerging field which leverages the decreasing cost of processing and memory to

---

\*This work was sponsored by the Defense Advanced Research Projects Agency issued by the AFRL under contract F30602-99-C-0131.

add intelligence in network nodes (routers and switches) to provide enhanced services within the network [30, 8]. The discipline of active networking can be divided into two sub-fields: Strong and Moderate AN. In Strong AN, users inject program carrying *capsules* into the network to be executed in the switches and routers. In Moderate AN, network providers provision code into the routers to be executed as needed. This code can provide new network based services, such as active caching and congestion control, serve as a mechanism for rapidly deploying new protocol versions, and provide a mechanism to monitor, control, and manage networks. We believe that moderate AN is the more compelling application of the technology in the near future, and the work described in this paper falls into this space. Note, however, that a provider can provision an interpreter which executes user code, providing a Strong AN service in a Moderate AN context. The OpenArch and OpenSig forums aim to provide, respectively, open network architectures and signaling interfaces, which are necessary but not sufficient to provide AN.

Active Networking is related to IN (intelligent networking), which provides intelligence and service creation mechanisms in the PSTN (public switched telephone network). While the goals of IN and Moderate AN are similar, the architecture of the PSTN and telephony switches resulted in IN protocols and mechanisms which are far too restricted, and inappropriate for use in the general data networking context with intelligent end systems.

Active networking can provide intelligence in the context of any layer 3 protocol. While the design and implementation described in this paper are done within the context of IP, the high level architectural principles apply to any networking technology, including IPv4, IPv6, and ATM.

## 1.2 Active Node Architecture

The key component enabling active networking is the *active node*, which is a router or switch containing the capabilities to perform active network processing. The architecture of an active node is shown in Figure 1, based on the DARPA active node reference architecture [7].

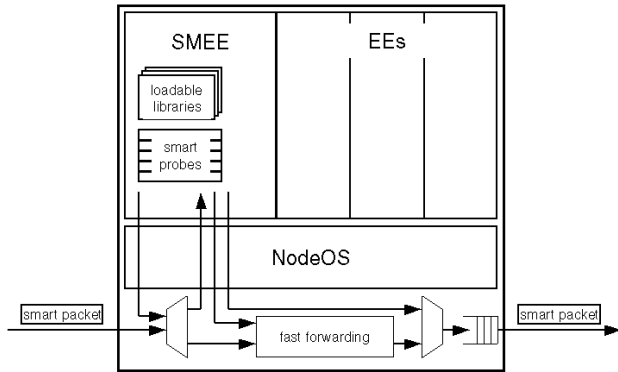


Figure 1. Architecture of an Active Node.

The conventional node hardware and software provide the basic forwarding, routing, and signaling capability of the node. Packets typically pass through the normal forwarding fast path of the node. Packet filters detect *smart packets* that need active processing, which are demultiplexed and sent up the active path. The NodeOS provides operating system services for the active node. Execution environments (EEs) provide the language and execution environment for active code. Active applications (AAs) execute to provide active services.

The SENCOMM management EE (SMEE) is an EE that provides an environment for the execution, monitoring, and control of AAs for node management (MAAs). MAAs consist of *smart probes* and *loadable libraries*, which will be discussed in detail in Section III. For now, it is sufficient to note that loadable libraries consist of code and data that is likely to be shared and reused among multiple MAAs, and smart probes are monitoring and control programs dynamically installed by smart packets.

The standard node reference model does not capture the details of high performance router and switch platforms, which have a distributed hardware architecture rather than a single central processor and memory. In particular, the forwarding function is implemented in hardware as a switch fabric, with packet processing distributed across the input and output interfaces. Thus, the location of active service functionality and implementation of active datapaths is more complex, and must be distributed among the *per port* processing functions [13, 29]. Monitoring and control MAAs, however, will be located in the control processor of

a high performance router, even though the packet filters are distributed.

## 2 Philosophy and requirements

By building a network control, monitoring, and management environment on active networks technology, we provide a flexible control and monitoring service that is capable of tracking networking technologies and new service offerings. However, this flexibility may result in multiple solutions, each with varied costs that impedes the robustness of the management infrastructure. To minimize this uncertainty, we believe a conservative design philosophy for active management is warranted. Furthermore, even a conservative design philosophy imposes requirements on the basic services needed from an active node for active management. Understanding the implications of a conservative design approach will help measure the merits of less conservative approaches.

### 2.1 General monitoring and management goals

We propose a set of goals for monitoring and managing nodes using active network technology. First, the development of management tools using this technology is focused on monitoring and managing both conventional and active networks. The management tools developed should be able to monitor an active node or use an active node to monitor one or more conventional nodes. Second, the management environment must be able to dynamically change or update the network management modules in current use. The capability to change or update modules in use permits the application to track current developments in the network, while continuing its previous role. Third, the management applications deployed in the environment must be able to monitor and control the management environment itself.

### 2.2 Design philosophy

We espouse a conservative design philosophy for active monitoring, management, and control, which allows realistic insertion of the technology and robust operation in the presence of the conditions which require network control and management.

Active networking technology need not be deployed in every network node to provide active services, in fact requiring wholesale replacement of networking infrastructure is not a realistic deployment scenario. Thus we support networks in which only some of the nodes are active, which we call *selective embedding* [29]. In the case of active management of conventional networks, active nodes are scattered throughout the network. As the ratio of active to passive

nodes increases, so does the fidelity of monitoring and ability to exert active control and management. Furthermore, the architecture should support a range of dissimilar instruction sets and hardware architectures for active nodes.

Since the goal of this work is to monitor, control, and manage both active node and networks (passive and active) we believe it is essential to depend on the presence of minimal network functionality, in particular, only the basic network layer protocol, in our prototype IP. Since actions taken by active management may be in response to network failures, there should be minimal dependence on network services and servers, such as DNS. When such services are available, operation is enhanced, but the basic functionality should not depend on them.

Note that while the SENCOMM architectural instantiation and implementation are based on IP, there is nothing fundamental in the design principles or high level architecture that requires this. SENCOMM could operate using any network layer protocol, and in fact could operate in an active network that supports multiple simultaneous network layers.

### 2.3 Basic requirements

We have identified a basic set of requirements that the active nodes must provide to adequately monitor, control, and manage the network, and to support the operation of a management EE. These requirements have been used as a basis for the SENCOMM architecture. In some cases the SENCOMM implementation is limited by the capability of current node capabilities, in particular, that ability to arbitrarily filter/reinject packets from/to the node forwarding path.

**Filtering** – While some packets may be explicitly tagged to be passed to the management EE, we need the general ability to monitor or extract packets from the normal forwarding path. The packet filter performing this function must be capable of matching on network layer header fields, or on an arbitrary match of the payload (note that payload encryption limits the ability do do this). For example, the management EE may need to monitor packets associated with a particular IP flow, TCP connection, or application UDP port binding (the latter two within the IP payload). In some cases these may be fields to which a format template may be applied, but in other cases an arbitrary pattern match through the entire packet may be needed. Furthermore, the management EE must be able to either grab a copy of a packet (for monitoring) or intercept the packet before other processing takes place.

**Active Datapath** – The active node must be able to support an active datapath that routes packets to AAs through

the corresponding EE. Generally, this consists of filtering and demultiplexing active packets from the normal forwarding path, and reinjecting them into the normal path after processing. Packets may be reinjected before the forwarding process or delivered to a particular output port, bypassing native routing functions. Packets may also be sunk or generated by the AA. In the case of the management EE, richer semantics are needed. In particular, we believe the MEE should have first access to incoming packets, and be able to inspect or modify them before other EEs.

**MIB Access** – The active node must provide access to MIB data, however the data may be provided by means other than SNMP.

**Secure Management** – The active node must control access to resources such as CPU scheduling, virtual memory, bandwidth, raw interfaces, and management information. Additionally, it must provide services for authenticating and encrypting communications as well as verifying management protocol messages.

**Local Management** – To enable the NodeOS or non-management EEs to be managed and controlled by the SMEE, they must export a management interface, such as a MIB or an API.

**Persistent Storage** – The active node should provide persistent storage for bootstrapping the SMEE and for long term storage of management code. If the management EE is booted from the storage, the persistent storage must be secure. Both secure bootstrapping protocols and secure persistent storage architectures exist in the literature [4, 25].

**Distributed Time Service** – The active node should provide a distributed service for accurate time (such as NTP [18]) so that probes on different active nodes can accurately timestamp data or make a configuration action at the same time. For robustness, distributed time service is *not* required for proper operation of the underlying SMEE. We believe a coarse level of time accuracy will be available to support other services on the node, e.g. security, however the SMEE can take advantage of a finer degree of accuracy than is required these other services.

## 3 SENCOMM Operational Overview and Scenarios

In this section we introduce the role of SENCOMM from a network-wide view and introduce some operational scenarios and examples. SENCOMM resides on an arbitrary

set of active nodes in the network. As indicated previously, the SENCOMM architecture supports a selectively embedded active network architecture in which not all nodes need to be active, nor do all active nodes need to run SENCOMM. We expect, however, that to gain the maximum benefit, active nodes are likely to run SENCOMM.

SENCOMM management EEs (SMEEs) reside on SENCOMM active nodes, and exchange active probes with one another to control, monitor, and manage the network as a whole, and active nodes in particular. Probes can be injected into the network from a management workstation, or may be created automatically by individual SMEEs. Probes may reside in SMEEs for a period of time, and use the services of SENCOMM loadable libraries, which contain stable data and serve as a commonly used active code base.

As an example scenario, a network management workstation injects a monitoring probe into the network. The probe is multicast to the set of SENCOMM nodes, and resides within each SMEE, monitoring active node and link state, including active processing load and traffic characteristics. The probes maintain both average and minimax statistics, and contain trigger thresholds that invoke alarm conditions. In the case of periodic statistics, information is returned to the management workstation via concast [6] (reverse multicast). Data fusion occurs at each merge point (average or minimax). In the case of alarms, a probe is sent directly to the management workstation indicating the alarm. If the alarm is able to automatically generate corrective management action a probe is dispatched for this purpose (for example routing table updates to load balance in response to building queues or a saturated link).

### 3.1 An example probe

In order to demonstrate the general SENCOMM operational framework, we describe the operation of a simple probe that performs a SNMPv1 `get`. More detailed descriptions of existing SENCOMM probes are described in [11].

To run a probe in the network, two separate Java programs, a launching application and the probe are needed. The launching application, appropriately named the launcher, prepares the probe so that it can be transported in the network. The probe is the code that actually executes in the SMEE.

In the following command:

```
> java smaas.snmp.SendSnmpGet [hostname] [OID]
```

`SendSnmpGet` launches the `SnmpGet` probe that returns one or multiple SNMP object identifiers, or OIDs, on the target host, identified by `hostname`. `SnmpGet` is implemented using classes from the AdventNet Java SNMP APIv3 [2]. Optionally, `SendSnmpGet` can take a `-p` argument to specify the SNMP port and `-c` argument for the

community string. The sources for `SendSnmpGet` and `SnmpGet` can be found in the Appendix.

`SendSnmpGet`, which extends the `SendProbe` class of the SENCOMM API, assembles the `SnmpGet` code to be placed in the packet, collects initialization data for the program, creates the Probe message payload, generates an ANEP packet, encapsulates the ANEP packet in a transport protocol packet, launches the packet into the network, monitors for messages from the probe, and displays the resulting data.

`SnmpGet`, which extends the `SmartProbe` class, requires the existence of the AdventNet SNMP API library for operation. For this example, assume that the library is resident and does not need to be fetched. Upon arrival, the SMEE saves the name of the probe for future reference and verifies that the probe language is supported. It then extracts the probe code from the Probe message, loads the probe code into a distinct namespace, and initializes the probe with its name. The SMEE then loads any libraries the code requires, i.e. the SNMP API, and resolves any missing classes or dependencies. The SMEE executes the probe, which issues one or more SNMP `gets` to the local SNMP daemon, and returns the data to the originating node in a Data message. Upon probe exit, the SMEE reclaims any resources used by the probe.

## 4 SENCOMM Architecture and Design

By building SENCOMM on active networks technology, we provide a flexible control and monitoring service that is necessary to track networking technologies and new service offerings. This approach has three advantages. First, the information content returned to the management center can be tailored (in real-time) to the current interests of the management center, thus reducing the back traffic as well as the amount of data requiring examination. Second, many of the management rules employed at the management center can now be embodied in programs which, when sent to managed nodes, automatically identify and potentially correct problems without requiring further intervention from the management center. Third, the monitoring and control loop is shortened – measurements and control operations are taken during a single packet's traversal of the network, rather than through a series of `set` and `get` operations from a management station.

More detailed documentation of the SENCOMM architecture and subsequent design can be found in [15] and [16], respectively.

## 4.1 SENCOMM Management Execution Environment (SMEE)

The SMEE has two primary functions: providing an execution environment for smart probes engaged in performing management activities for the active network and helping the NodeOS manage the active node itself, including other EEs and the NodeOS. The SMEE may be implemented as either a privileged active application inside an EE or as an execution environment itself in the active networks framework, as long as it is sufficiently authorized to access all the node operations that need to be managed.

There are several aspects of this environment that deserve to be explored in some detail. The SMEE must provide data isolation between running applications, and prevent data crossing the protection domain between applications. The SMEE must isolate the smart probes from each other. However, it must allow a smart probe to explicitly make selected data available to other probes to support a variety of management scenarios. For example, a probe can leave persistent data to be used by probes currently running on the system or for probes that follow it.

Sharing data also permits a hierarchical partition among probes. For example, a set of monitoring probes offer the data they collect to a set of control probes on same the platform. Both the monitor and control probes, when partitioned, are simpler to implement and easier to deploy.

Another service provided by the SMEE is support for loadable libraries (Section 4.5). The SMEE is responsible for finding, retrieving, and caching libraries for smart probes. This allows the SMEE to help the probes amortize the cost of finding and retrieving the library code over all the probes that use that code. Additionally, the SMEE can use libraries to dynamically extend its environment, similar to the way in which kernel modules can be used to dynamically extend a FreeBSD or Linux kernel. These libraries can be used by any of the probes that run in the SMEE.

The SENCOMM loadable libraries can be used to adapt a SMEE to the node on which it is running, to allow it to monitor the node itself. Libraries that contain the functions necessary to monitor the particular NodeOS and EEs running on the node can be loaded upon SMEE initialization to customize it for that node. Additionally, as new EEs are started on the node, the SMEE can add libraries to monitor the new EEs.

## 4.2 SENCOMM Messaging

SENCOMM Probes are mobile executable programs that perform management functions. The probes send messages to each other and to management stations. SENCOMM messages are encapsulated in the Active Network Encapsulation Protocol (ANEP). ANEP, along with a transport

protocol, is a mechanism for transporting Active Network messages over an IP network infrastructure. The ANEP header includes fields that identify the execution environment of the message and define the forwarding semantics if the environment is not available. Optional fields in the ANEP header specify source and destination addresses and security

To send a Probe packet, the active node prepares a SENCOMM probe message which contains the code and a unique global identifier that identifies this probe in the network. This message is encapsulated in an ANEP packet, which is then encapsulated in a transport protocol packet, usually UDP, with a specified port number, and addressed to the recipient.

The receiver waits for ANEP packets to arrive, i.e., UDP packets sent to a particular port. The ANEP header is processed and the payload is sent to the specified execution environment. ANEP header processing can optionally include security services such as authentication and data integrity. If any of the security checks fail, the packet is discarded.

### 4.2.1 Fragmentation and Reliable Delivery of SENCOMM Packets

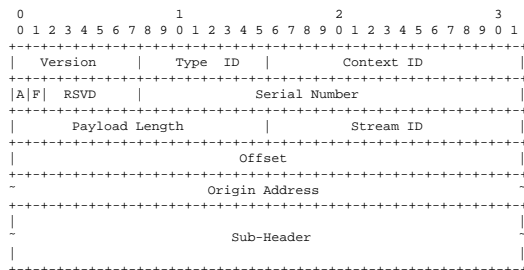
If any SENCOMM protocol packet is longer than 64 kilobytes it must be fragmented into multiple packets (see Appendix). In practice, the packets need to be fragmented into smaller than 64 kilobyte segments, because at the SENCOMM layer it is impossible to determine the exact length of the total message since the ANEP header does not have a fixed size. The length of the SENCOMM portion of the packet must have a maximum length which is small enough to allow for all other protocol headers.

Since SENCOMM probes are not restricted to fitting into a single datagram, it is necessary to ensure that they are delivered reliably. Recalling our conservative design philosophy, control messaging should have as simple a state machine as possible. Furthermore the mechanism chosen should be as lightweight as possible, since probes may have to be injected into the network when it is congested and the reliability mechanism should not cause undue problems with probe delivery. Early releases of SENCOMM implement a simple reliability mechanism on top of UDP; the preferred reliable transport mechanism for later releases is RDP [26]. It offers an appropriate balance between reliable message delivery while keeping messaging state at a minimum.

## 4.3 SENCOMM Packet Format

The SENCOMM protocol allows SMEEs running on different nodes to communicate with each other and with management clients. A SENCOMM packet is encapsulated

in an Active Network Encapsulation Protocol (ANEP) [3] datagram, which may be transported using UDP/IP, TCP/IP, or RDP/IP. All the packet types have a common header (Figure 2).



**Figure 2. The SENCOMM Common Header.**

The SENCOMM protocol header has eleven fields: version number, type, context, address type flag, more fragments flag, serial number, length, stream ID, offset and origin address. The `Version` number is used to identify language upgrades and packet format changes. The `Type ID` field indicates which of five types of SENCOMM protocol packets is contained in the message: executable code, a loadable library, data, or requests for a certificate or library. Probe packets carry executable code along with data that is used as input to the probe. The data is mutable so each node executing the probe may have different inputs to the probe. Library packets carry loadable libraries that may be sent in response to a library query packet, which requests the library by name and a range of acceptable version numbers. Message packets carry data, status, and error messages from a smart probe to a specified network management device. Messages also carry certificates in response to a certificate query message that requests a certificate by identifying the principle and certificate authority. Library queries identify libraries or other data the probe needs to operate.

The `Context ID` field holds a value that identifies the originator of the probe. The context value is uniquely generated for each client, and is unique for that client within that host. The value is placed into outgoing Probe packets. As Probe packets traverse the network and generate one or more responses (Message packets), the context value is used to identify the client to which the responses must be delivered.

The `A` flag is used to indicate whether the `Origin Address` is of type IPv4 or IPv6.

The `F` flag is used to indicate whether or not more fragments for this `Stream ID` are coming.

The `Serial Number` field holds a value that is used to identify the collection of instances of this smart probe. The number must be unique within the management context on the origin, as a management application can have

several smart probes active in the network. This value allows a client to match response packets with injected programs. Like the context field value, response packets echo the serial number value of the Probe packet.

The `Payload Length` field holds the total length of SENCOMM message, including the common header.

The `Stream ID` field holds a unique, strictly increasing number that, along with the probe name triple and IP source address, identifies a set of fragments as belonging to the same packet.

The `Offset` field contains an offset, in 64 bit words, of the first byte in this message fragment from the start of the SENCOMM message.

The `Origin Address` field holds the address of the node that injected the probe into the network. This address is the default address for status and error messages from a running probe.

The triple of `Origin Address`, `Context ID`, and `Serial Number` constitutes the *name* of a smart probe. A probe may be executed at more than one node in the network. Yet, while copies of the probe are being forwarded, each instance of the probe maintains the same name. A new probe, even if it originated at the same host and contains the same code will have a different name, determined at the probe's launch into the network.

The triple allows messages from a smart probe to be delivered to the correct management process that issued the probe as well as responses to certificate and library requests to be correctly delivered. The issuing process can distinguish the messages sent by different instances of the same probe by the nodes' addresses that sent the responses.

The SENCOMM Protocol depends upon ANEP to provide the necessary authentication services. A set of security options has been proposed [22, 24] for both hop-by-hop and stream active networks communications where portions of the packet may change at each hop. The proposed options, `Hop Integrity`, `Credential Authenticators`, `Credential Fields`, `In-line Policy`, `Static Payload` and `Varying Payload`, would allow the verification of data integrity of the static portion of the packet and hop-by-hop verification of the changes in the variable section of a packet. The options also permit the transport of both multiple signatures and multiple sources of credentials for the verification of multiple signatures.

#### 4.4 Smart Probes

Smart probes are executable programs that perform management functions. A smart probe is delivered to the active router inside a SENCOMM Probe packet. While the executable program is immutable, i.e., it does not change from hop to hop as the probe traverses the network, there is a provision for mutable probe initialization data.

The original requirements for a smart probe were few. The probe should have a globally unique name, so that any probe can be distinguished, and thus controlled, in the network. It should be able to continue functioning on the node after the packet that carried it to the node has been forwarded, permitting the efficient dissemination of probes in the network. It should be able to access functions native to the SMEE or those that reside in an external library, allowing the probe to re-use frequently accessed functions, amortizing the cost of transporting those functions to the node over multiple invocations, and reduce the size and complexity of the code in the probe. It should be able to register for an event and time out if it does not occur.

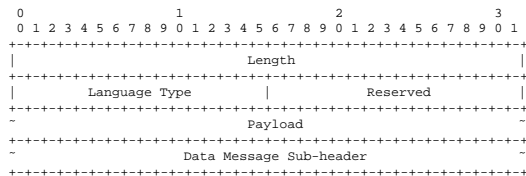


Figure 3. The Probe Sub-header.

The Probe sub-header contains fields that identify the length of the executable code and the code type. The Data message sub-header provides initialization data for the probe. The structure of the data is defined by the probe. Global probe name uniqueness is addressed by the triple [Origin Address, Context ID, Serial Number] from the SENCOMM common header, described in Section 4.3. Any probe and all its copies can be uniquely identified and controlled in the network. The probe interface to libraries is described in Section 4.5. The programming language and EE provide the mechanisms for a probe to continue functioning after the packet that carried it there moves on and support callback and timer programming interfaces [9].

#### 4.4.1 Probe Initialization Data

Securely sending data to be used as input to a probe when it is executed is a difficult problem [24]. The data must be sent along with the probe, so there is no chance of the two being separated or arriving in the wrong order. On the other hand, there are authentication and data integrity issues, since the initialization data can change from hop to hop, while the probe is static. While the probe portion of the message should have an authenticator added by the originating node and verifiable by any evaluating node, the changing data can only have an authenticator added on a hop-by-hop basis by those nodes making the changes. Recent work in the area of active networks security addresses this issue [24].

## 4.5 Loadable Libraries

Management libraries differ from smart probes in that they are not executable pieces of code in their own right. The libraries must be invoked by a smart probe to be executed. They are designed to provide classes, methods, and data structures to be used by one or more smart probes. This allows common methods to be shared among multiple smart probes, so that each probe does not have to re-implement the contents of the library, reducing the size and complexity of the probes.

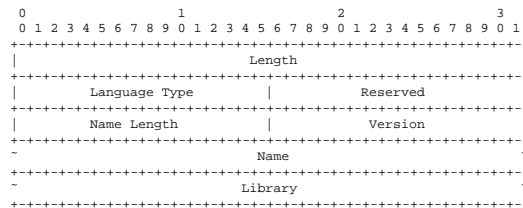


Figure 4. The Library Sub-header.

The library sub-header contains fields that identify the length, type, name and version of the library contained. The types of files that can be transported are Java class and .jar files, binary, and text.

Every library has a unique name so that smart probes and other libraries can unambiguously refer to the library by name. SENCOMM names libraries using the proposed Uniform Resource Name (URN) syntax [19], for example `urn:X-an-ll://sencomm.bbn.com/BBN-concast-library/2`.

This example name includes the authority that named the library, `sencomm.bbn.com`, a string that identifies the library, `BBN-concast-library`, and optionally a version number, `2`. The syntax also allows the library to be named without a particular version number to refer to any version of the library.

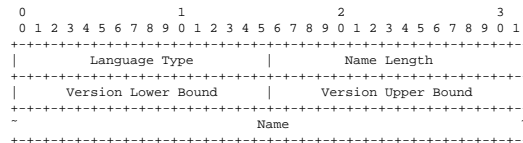


Figure 5. The Library Query Sub-header.

Libraries are loaded when the SMEE is started or as needed by smart probes and other libraries. The library fetching mechanism attempts to find the library first in a local cache, then from a configured local library path, then from servers specified by the AA when requiring the library, and, as a last resort, from configured default library servers.

Any required libraries that are not currently loaded are fetched via a library query message and dynamically loaded before executing the probe. The required libraries are specified by their name and one or more acceptable version numbers. The query identifies which version of the library it requires by specifying either a particular version number of the library, any version of the library greater than or less than a particular version number, or any version of the required library. The latter two specifications leave the selection of the library to the recipient of the query.

## 5 Implementation Environment and Issues

Successful implementation of the SENCOMM design requires support from multiple components, including the development language, the EE, and other components of the Active Networks Node Architecture. This section details a selection of the issues and the resulting decisions that were made during the implementation of SENCOMM v0.81.

### 5.1 Implementation and Development Environment

We evaluated several different execution environments against the architecture discussed in Section 4. We found that the Active Signaling Protocol (ASP) EE from USC/ISI [5], which supports the development of control protocols, to be well matched to our needs and requirements.

The initial deployment of SENCOMM is on the ABone, the DARPA Active Networks Testbed [1]. ABone nodes use a custom daemon called `anetd` [28, 12], which performs the demultiplexing, EE loading and packet filtering functions. The NodeOS will support these functions, but at this time a NodeOS has not yet been deployed in the ABone. Additionally, the ASP EE is one of a small set of permanently available EEs in the testbed, minimizing another barrier to deployment in the ABone. ASP v1.4 supports SENCOMM without any additional modifications. Currently, SENCOMM v0.81 runs on FreeBSD 3.x-RELEASE and 4.x-RELEASE, and Linux 2.2.x and 2.4.x. The supported Java development environment is `jdk1.2.2`. Detailed information on configuring an ABone node to run the SMEE can be found in the SENCOMM documentation [10]. NodeOS support will be integrated into a future SENCOMM version.

### 5.2 Implementation Issues

The implementation environment was selected to minimize any difficulties and potential incompatibilities that might occur while implementing the design. Nevertheless, we expected the unique combination of implementing a network control and management system in an EE that was

not designed specifically for the purpose to require compromises with the SENCOMM design goals, as well as modifications to the chosen EE. The following subsections discuss the implementation issues uncovered in integrating SENCOMM into ASP and the decisions made to resolve them.

#### 5.2.1 SMEE Implementation: How SMEE differs from an ASP AA

As mentioned in Section 4.1, the SMEE can either be implemented as a standalone execution environment or as an AA in another EE, as long as it is sufficiently authorized to access all the node operation that need to be managed. We chose to implement the SMEE as a privileged AA that runs in the ASP EE [5], which minimizes the modifications to the base ASP distribution, allowing us to closely track ASP development and quickly include newly released ASP services within SENCOMM.

#### 5.2.2 SMEE Privileges

The ASP EE was designed to provide a high degree of protection against boundary violations by general unprivileged AAs. The ASP EE mechanisms for preventing boundary violations are based on Java's strong typing and safe pointer variables, which prevent an AA from accessing classes that it does not have a reference. For those portions of the EE that require dynamic protection, the ASP EE installs a custom Security Manager to block illegal accesses. ASP permits the definition of AA *capabilities* that define privileges.

Implementing the SMEE as a privileged ASP AA required defining the capabilities to permit the SMEE to run without incurring access violations from the ASP Security Manager. The SMEE has the capability to modify the EE's routing and interface tables. To enable it to issue arbitrary packet filters, it has the capability to perform arbitrary packet interception. In order for the SMEE to control its own probes, it has the capability to terminate AAs. The SMEE is allowed to use JDK sockets that are needed by the Java SNMP API [2] we are using. This SNMP API also uses Java threads which are normally prohibited from AA use. ASP uses a Java byte code re-writer in the ASP class loader that changes references of Java threads to ASP threads.

#### 5.2.3 MAA Class loaders

While the SMEE is an ASP AA, it differs from ASP in how it loads active code. For ASP, the loadable code is the AA, refereed to by an AASpec which contains a search path specifying one or more code servers. For SENCOMM, the probe carries the *main* function. One or more library references in the probe are used to resolve class dependencies. ASP currently uses a single class loader to load it's AAs.

However, using a single class loader does not permit the isolation between probes that SENCOMM requires to allow per probe granularity in probe creation and destruction. SENCOMM implements a class loader per-probe, thus enforcing isolation between probes and allowing both multiple probes use the same library without interaction and individual probes to use different versions of the same library.

#### 5.2.4 Multiple TypeIDs

SENCOMM has been assigned ANEP TypeID 25. It is required that both `anetd` and ASP can recognize that ANEP messages with ID 25 need to be passed to the ASP EE on which the SMEE is running. While this is permitted by the AN architecture, `anetd` currently has a one-to-one mapping of port to TypeID, such that the ASP EE's TypeID are the only packets that are passed up to ASP. ASP, similarly, does not support an AA using a TypeID different from its own, so it cannot recognize that it should process those packets or that it should send them to the SMEE.

To work around these problems, the current implementation configures ASP to listen on a separate UDP port for SENCOMM protocol messages, which are then delivered directly to the SMEE. The SMEE then executes its own ANEP protocol processing internally.

The ABone network I/O daemon, `netiod`, which replaces the network demultiplexing function of `anetd`, should permit the SMEE to register for TypeID 25 packets and remove the need for internal ANEP protocol processing. At this time, we are planning to experiment with a beta version of `netiod` when it becomes available.

#### 5.2.5 Security

Security is an important aspect of active networking. It is a concern to protect communications in the SENCOMM protocol, as well as to control access to the management resources on the active node. The SENCOMM project is not focusing on implementing security services for the SMEE. Instead, SENCOMM design is targeted to be compatible with other security efforts in the active networks research community [23, 21, 20]. However, the lack of currently available security services has influenced some design and implementation decisions.

For example, recall the previously discussed issues on the secure transmission of the initializing data for a probe. The probe message was designed to separate the mutable and immutable parts of the message. The data is placed in the mutable area of the packet and the probe in the immutable part. Had ANEP security services been implemented and there were different signatures for mutable and immutable fields, we would have designed the message to include the additional fields for the security services that

would allow both end-to-end authentication and data integrity of the probe portion and hop-by-hop authentication and data integrity of the initialization data. However, since the security services for communications between active nodes was still an open research area and thus not implemented, the solution chosen for the current implementation was to add the data to the end of a probe packet, which separates the two, but does not allow insertion of security service fields between the two.

#### 5.2.6 NodeOS Management Interfaces and Filtering Semantics

In order to properly manage the active node, the SENCOMM must have access to management variables and functions in the NodeOS. The NodeOS must provide management hooks and an API to monitor and adjust those hooks, however the current NodeOS specification [27] and current NodeOS implementations do not provide the necessary management information. `Anetd`, which currently provides some NodeOS services to active nodes on the active networks backbone (ABone), also does not provide any management API. We are working with both NodeOS designers and the `anetd` designers to include management services in the future.

Section 2.3 describes SENCOMM's requirements for a flexible packet filter that has the ability to copy the filtered packet to the SENCOMM and the ability to grab packets at different points in the input path. However, current NodeOS implementations do not provide complete filtering and re-insertion services. The packet filters commonly used, such as BPF [17] and DPF [14], only allow one application to filter each packet, even if multiple applications request it. Additionally, current NodeOSes only allow filtering at one point in the data path. These deficiencies limit SENCOMM's ability to fully monitor and control, and will need to be addressed in future active node and NodeOS architectures.

## 6 Status and Future Work

The SENCOMM project has implemented the SMEE and a set of MAAs for network management. Currently, there is an environment to launch, receive, execute and destroy smart probes. Many types of smart probes have been implemented which perform a range of functions from a simple RemoteLivenessTest (a.k.a `ping`) that demonstrates basic probe functions to probes that monitor the deployed SMEEs in the ABone.

The SMEE, which provides the environment for executing a smart probe, is implemented as a privileged ASP AA. The SMEE implements the SENCOMM and ANEP protocol processing state machines. The SMEE instantiates

probes, keeps track of all running probes, removes probes, and collects statistics on probes.

The SMEE instantiates a single class loader per smart probe, which allows the namespaces among probes to be kept separate. Besides enforcing probe isolation, separate namespaces allow multiple probes to use different versions of the same library without conflict.

The SENCOMM API provides a mechanism to allow probes to specify loadable libraries using the proposed Uniform Resource Name (URN) syntax. Loadable libraries consist of code and data that is likely to be shared and reused among multiple probes. Queries to search for and return libraries according to the supplied URN is included in the SENCOMM protocol. Recursive library loading, when libraries require other libraries, has also been implemented and tested.

## 6.1 Future work

While the project continues to develop applications to be deployed on the ABone, we are investigating how to use SENCOMM probes in an expanding set of roles. One of these roles is using active nodes and probes to monitor conventional nodes. An active node can host probes that check router configuration, observe for deviations from usual traffic patterns and perform long term baseline studies for future reference. Probes can also be directed to monitor file or application servers, reporting on the number of clients, processing load and traffic characteristics. The data could be used as input to algorithms to balance the request load to a pool of servers.

We are also using our experience with SENCOMM to develop, along with others in the active networks community, a reference architecture document for active networks management. The goal of the document is to outline a framework for the management of active networks and the active management of conventional networks. It will establish a set of management requirements for all components of the active node architecture and integrate the appropriate specifications from the current set of architecture documents.

## 7 Summary

In this paper, we presented the architecture, design, and implementation of SENCOMM, a Smart Environment for Network Control, Monitoring and Management. The SENCOMM Management Execution Environment (SMEE) was developed for node monitoring, control and management. It coexists with other execution environments and, given the proper interfaces from the NodeOS, can monitor and control them. Management applications, called smart probes, run in the SMEE. Probes can be injected into the network from a

management workstation, or may be created automatically by individual SMEEs. Probes may reside in SMEEs for a period of time, and use the services of SENCOMM loadable libraries, which contain stable data and serve as a commonly used active code base. SENCOMM protocol messages, including probe and library transport, are delivered to the active node within an Active Network Encapsulation Protocol datagram.

Our architecture is designed to actively control, monitor, and manage both conventional and active networks, and be incrementally deployed in existing networks. We presented a set of goals, a design philosophy, and a set of basic requirements for controlling, monitoring, and managing networks using active network technology. We discussed the operation and components of SENCOMM: the management EE, a messaging protocol, smart probes, and loadable libraries. We discussed the implementation issues uncovered in integrating SENCOMM into a selected EE and the decisions made to resolve them. We concluded with an overview of the current status of the work and indicate the directions of our future research.

While much of the experience gained from this project has similarities with other research prototypes, we have learned some lessons that should prove useful to the active networking research community.

Getting the the right management hooks in all the components to be managed early in the design cycle is difficult, without having an absurdly and unwieldy large set of hooks and API. It is important that this be an integrative process by the research community before deployment in production networks is considered. This problem is possibly as difficult as properly integrating security into the design cycle early.

It is clear that arbitrarily complex AAs are possible, given a sufficiently rich EE programming model. However, active networking requires significant limits to the programming model to provide sufficient security and resource bounds to protect the node and other traffic from misbehaving AAs. Management applications, however, need more functionality to monitor and control node resources, as well as other AAs and EEs (which are generally *not* allowed access to one another. Operating systems have access to instructions and hardware resources (e.g. I/O, scheduling, and memory management) from which user mode instructions are restricted. It appears that an analogous concept of privilege/restriction between management and non-management AAs is a useful distinction that should be supported by the NodeOS and EEs.

A specific example of this dichotomy that was an obstacle to SENCOMM deployment is that some EEs provide an overlay virtual network topology that obscures the physical network topology and restricts access to the routing algorithms and forwarding tables. While this has useful

properties for many AAs, it is unduly restrictive to management applications that need to control the physical links over which probes pass and directly manipulate node resources such as forwarding tables.

The SENCOMM project is focused on an active monitoring and control infrastructure that can be used to manage networks. Network-wide management is a significantly more difficult problem than node management. While we believe that we have provided useful tools to be used in the management community, it is important to understand that network management in general, and active network management in particular, is a long way from being a solved problem.

## Acknowledgments

We thank Joel Levin and David Waitzman for their help defining the SENCOMM architecture and initial design. We also thank Jay Lepreau, Pat Tullmann, Mike Hibler, Tim Stack, Eric Eide, Kristin Wright, Bob Braden, Steve Berson, Ted Faber, Bob Lindell, and Ken Calvert for their insightful discussions about managing the NodeOS, ASP, and anetd that have helped us refine our design. We thank Leonid Poutievski for his help in bringing the SENCOMM design into reality and for acting as an ambassador from the ActiveCast group at the University of Kentucky.

## References

- [1] ABone home page. <http://www.isi.edu/abone/>.
- [2] AdventNet. AdventNet SNMP Release. <http://www.adventnet.com/products/snmpbeans/>.
- [3] D. Alexander, B. Braden, C. Gunter, A. Jackson, G. Minden, and D. Wetherall. Active Network Encapsulation Protocol (ANEP). Technical report, Active Networks Group, July 1997.
- [4] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A secure active network architecture: Realization in switchware. *IEEE Network*, 12(3), 1998.
- [5] B. Braden, A. Corpa, T. Faber, B. Lindell, G. Phillips, and J. Kann. ASP EE: An Active Execution Environment for Network Control Protocols. Technical report, 1999. <http://www.isi.edu/active-signal/ARP>.
- [6] K. Calvert. ConCast, 1999. <http://www.dcs.uky.edu/~acast/concast.html>.
- [7] K. Calvert, ed. Architectural Framework for Active Networks. AN draft, AN Architecture Working Group, 1998.
- [8] K. L. Calvert, S. Bhattacharjee, E. Zegura, and J. P. Sterbenz. Directions in active networks. *IEEE Communications Magazine*, 36(10), Oct. 1998.
- [9] M. N. Condell and R. R. Hain. SENCOMM Programmer's API, 2002. <http://www.ir.bbn.com/projects/sencomm/spapi.ps>.
- [10] M. N. Condell and R. R. Hain. User's Guide to the SENCOMM Environment in the ABONE, 2002. <http://www.ir.bbn.com/projects/sencomm/userguide.ps>.
- [11] M. N. Condell and R. R. Hain. Writing a Probe in the SENCOMM Environment, 2002. <http://www.ir.bbn.com/projects/sencomm/probeguide.ps>.
- [12] S. Dawson, M. Molteni, L. Ricciulli, and S. Tsui. User Guide to Anetd 1.6.3). Technical report, Sept. 2000. <http://www.csl.sri.com/activate/anetd/doc/anetd-user-guide.ps>.
- [13] D. Decasper, B. Plattner, G. Parulkar, S. Choi, J. DeHart, and T. Wolf. A scalable, high-performance active network node. *IEEE Network*, 13(1):8–19, Jan.-Feb. 1999.
- [14] D. Engler and M. F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proceedings of ACM SIGCOMM '96*, pages 53–59, Stanford, California, Aug. 1996.
- [15] A. W. Jackson, J. P. Sterbenz, M. N. Condell, J. Levin, and D. J. Waitzman. Sencomm architecture. Technical Report 1278, BBN Technologies, Verizon, 2001. <http://www.ir.bbn.com/projects/sencomm/doc/architecture.ps>.
- [16] A. W. Jackson, J. P. Sterbenz, M. N. Condell, J. Levin, and D. J. Waitzman. Sencomm design, 2001. <http://www.ir.bbn.com/projects/sencomm>.
- [17] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Technical Conference*, pages 259–269, San Diego, California, Jan. 1993.
- [18] D. Mills. Network time protocol (version 3) specification, implementation. Request for Comments 1305, Internet Engineering Task Force, Mar. 1992.
- [19] R. Moats. URN syntax. Request for Comments 2141, Internet Engineering Task Force, May 1997.
- [20] S. Murphy. Active node security architecture diagram. <ftp://ftp.tislabs.com/pub/activenets/secsrarchpicture.prm.ppt>.
- [21] S. Murphy. Proposed modifications to security architecture: Topics 1-5. Active Nets security mailing list. <http://www.ittc.ukansas.edu/Projects/ActiveNets/>, July 1999.
- [22] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *2001 IEEE Open Architectures and Network Programming*, pages 63–70, Apr. 2001.
- [23] S. Murphy, ed. Security Architecture for Active Networks. AN draft, AN Security Working Group, July 1998.
- [24] S. Murphy, ed. Security Architecture for Active Networks. AN draft, AN Security Working Group, May 2001.
- [25] D. Nagle. Network Attached Secure Disk. <http://www.pdl.cs.cmu.edu/NASD/>.
- [26] C. Partridge and R. Hinden. Version 2 of the Reliable Data Protocol (RDP). Request for Comments 1151, Internet Engineering Task Force, Apr. 1990.
- [27] L. Peterson, ed. NodeOS Interface Specification. AN draft, AN Node OS Working Group, Jan. 2000.
- [28] L. Ricciulli. Anetd: Active NETWORKS Daemon (v1.0). Technical report, Aug. 1998. <http://www.csl.sri.com/ancors/anetd/>.
- [29] J. P. Sterbenz. Intelligence in future broadband networks: Challenges and opportunities in high-speed active networking. In *Proceedings of the IEEE International Zurich Seminar on Broadband Communications (IZS 2002)*, Feb. 2002. (to appear).

[30] D. L. Tennenhouse and D. J. Wetherall. Towards and active network architecture. *ACM Computer Communication Review*, 26(2):5–18, Apr. 1996.

## A SENCOMM message processing

### A.1 Probe Processing

When sending a SENCOMM Probe, a node MUST:

1. Set `Version` to 1 and `TypeID` to 1.
2. If the probe is originating at this node:  
Set the `ContextID` appropriately for the management context and the `Serial Number` within that context. Also set the `Origin Address` to the IP address of the sending node and set the `A` bit to indicate the type of address.  
If the probe is not originating at this node:  
Use the `ContextID`, `Serial Number`, and `Origin Address` received with the probe from the origin.
3. Add the length, language in which the probe is written, and the probe itself in the probe sub-header.

When receiving a SENCOMM Probe for which it is the end-point, a node MUST:

1. Save `ContextID`, `Serial Number`, and `Origin Address` as the name of the probe in the `ProbeTable`.
2. Verify that the probe language is supported. If it is not, then it SHOULD return error status 1.
3. Extract the probe code
4. Load it into the environment, initializing it with its name.
5. Load any libraries the code requires
6. Execute the probe.

When receiving a SENCOMM Probe for which it is a transit node, a node MUST:

1. Save `ContextID`, `Serial Number`, and `Origin Address` as the name of the probe in the `ProbeTable`.
2. Verify that the probe language is supported. If it is not, then it should silently forward the probe by resending it to the destination address without changing the source IP address. Additionally, the node MAY return error status 1 to the originating node.
3. Extract the probe code

4. Load it into the environment, initializing it with its name.
5. Load any libraries the code requires
6. Execute the probe.

### A.2 Library query processing

When sending a SENCOMM Library Query, a node MUST:

1. Set `Version` to 1 and `TypeID` to 5.
2. Set the `ContextID`, `Serial Number`, and `Origin Address` to the name of the probe that is requesting the library.
3. Set `Language Type` to the language in which the requested library is written.
4. Set `Version Lower Bound` and `Version Upper Bound` fields to the minimum and maximum version numbers, inclusive, of the library which are acceptable.
5. Set the `Name` and `Name Length` to the name of the library requested and the length of the name.

When receiving a SENCOMM Library Query addressed to it, a node MUST:

1. Find a local copy of the library indicated in the request. The library MUST match the `Name` and `Language Type` indicated in the query and have a version number in the specified range.
2. If the library cannot be found, return a status message with an error code 3.
3. If the library is found, return a message with a library sub-header containing the requested library.

#### A.2.1 Library Processing

When sending a SENCOMM Library, a node MUST:

1. Set `Version` to 1 and `TypeID` to 2.
2. If the library is associated with a particular probe, set `ContextID`, `Serial Number`, and `Origin Address` to the name of the associated probe. If it is not associated with a probe, set the fields to zero.
3. Add the length, language in which the library is written, and the library itself in the library sub-header.

When receiving a SENCOMM Library addressed to it, a node MUST:

1. Save the `Name` and `Version` from the library subheader for the name and version of the library.
2. Verify that the library language is supported. If it is not, then return an error status 1 and discard the message.
3. Extract the library code
4. If the message is in response to a query and the probe that the library is addressed to is running locally, the library should be loaded into the probe.
5. The library may be saved to the local file system so it is available in the future.
6. Recursively load any additional libraries that are required.

## B SENCOMM message fragmentation protocol

When sending a message that is longer than `MAX_FRAGMENT_SIZE` where `MAX_FRAGMENT_SIZE < 64` kilobytes:

1. While the length of the remaining unsent part of the original message and the size of the SENCOMM common header is longer than `MAX_FRAGMENT_SIZE`:
  - (a) Create a new message of `MAX_FRAGMENT_SIZE` that contains a copy of the original SENCOMM common header and the next `MAX_FRAGMENT_SIZE - the size of the SENCOMM common header` bytes of the original message. The fragment, not including the header, must be a multiple of 8 bytes long. In the first fragment, the remaining bytes of the original message does not include the SENCOMM common header, but does include all subheaders.
  - (b) Set the `F` flag to indicate more fragments are coming.
  - (c) Record the length of this fragment (including the header) in the `Payload Length` field.
  - (d) Record the offset (in 64 byte words) from the beginning of the original message, not including the common header, in the `Offset` field.
2. For the last fragment:
  - (a) Create a new message that contains a copy of the original SENCOMM common header and the remaining bytes of the original message.
  - (b) Unset the `F` flag to indicate it is the last fragment.
  - (c) Record the length of this fragment (including the header) in the `Payload Length` field.
  - (d) Record the offset (in 64 byte words) from the beginning of the original message, not including the common header, in the `Offset` field.
3. Send all of the fragments.

Upon receiving a packet where either the `F` flag is set or the `Offset` is greater than 0, reassembly is required.

1. Check if an existing reassembly buffer exists to which this fragment belongs. The buffer can be identified by matching the probe name, source address of the packet, and `Stream ID`. If no matching buffer exists, then create one.
2. Set a timer that indicates the maximum amount of time the node will wait for all the fragments to arrive.
3. Add the new fragment to its reassembly buffer.
4. Check if all the fragments have been received. All fragments have been received, when the last fragment (`F` flag is unset) is present and there are no gaps between the first (`Offset = 0`) and last fragments. A gap can be detected if a fragment hasn't arrived which has an `Offset` equal to the sum of the `Offset` and `Payload Length` of the previous fragment.
5. If all the fragments are present, recreate the original message by creating a message that contains, in order, the common header and the data following the common header from each consecutive fragment. Remove the reassembly buffer.
6. If all the fragments are not present, check the timer. If it has not expired, wait for more fragments to arrive. If it has, remove the reassembly buffer.

## C SendSnmpGet source

```
package smaas.snmp;

import smee.SmartProbe;
import smaas.ParseOptions;
import smee.protocol.SMEEPacket;

import asp.inet.AddressIP;
import asp.inet.AddressIPv4;

import java.net.InetAddress;
import java.net.DatagramSocket;
import java.net.DatagramPacket;

...

public class SendSnmpGet extends smaas.SendProbe {
    private static final int COMMUNITY = 0;
    private static final int PORT = 1;

    // Take care of getting options
```

```

private static String usage =
    "SendSnmGet [-c community] [-p port] host OID ";
private static String options[] = { "-c", "-p"};
private static String values[] = { null, null};
private static ParseOptions opt;

/**
 ** convert command line arguments to byte array :
 ** here domain name is converted to IP Address
 ** and then to byte array
 **
 ** @param args    command line arguments
 **
 **/
protected static byte[] getData() {

StringBuffer joinedArgs;
int i = 0;
String defPort = "161";
String defCommunity = "public";

/**
 ** joinedArgs has the form:
 ** <port>:<community>:<destination>:<oid>
 **/

if (values[COMMUNITY] != null)
    // set the community if specified
    defCommunity = values[COMMUNITY];

if (values[PORT] != null)
    defPort = values[PORT];

// check if there are arguments for the probe
try {

    joinedArgs = new StringBuffer(defPort + ":" +
        defCommunity + ":" +
        opt.remArgs[0] );

    for( i = 1; i < opt.remArgs.length; i++) {
        joinedArgs.append(":");
        joinedArgs.append(opt.remArgs[i]);
    }

    return (joinedArgs.toString().getBytes());

}
catch (Exception e) {
    e.printStackTrace();
    return null;
};
}

/**
 ** convert command line arguments to byte array :
 ** here domain name is converted to IP Address
 ** and then to byte array
 **
 ** @param args    command line arguments
 **
 **/
public static void main(String[] args) {

InetAddress destAddr;
int retVal;
DatagramSocket sock = null;

System.out.println("args.length : " + args.length);

opt = new ParseOptions(args,options,values, usage);

// check for at least hostname and
// one OID in remaining arguments
if (opt.remArgs.length<2) opt.usage_error();

try {
    // there is method getByName on AddressIpv4,
    // but it's protected, so make

```

```

// call to java InetAddress class
destAddr = InetAddress.getByname(opt.remArgs[0] );
sock = new DatagramSocket();
if (sock == null) {
    System.out.println("Unable to open socket");
    System.exit(0);
}

retVal = launchProbe(loadCode("smaas.snmp.SnmGet"),
    getData(),
    destAddr, sock);
}
catch (java.net.UnknownHostException e) {
    e.printStackTrace();
    retVal = SmartProbe.SEND_ERR;
}
catch (java.net.SocketException s) {
    s.printStackTrace();
    retVal = SMEEPaket.SEND_ERR;
}

if (retVal != SmartProbe.SEND_OK) {
    System.out.println(
        "Could not launch the probe. Error : " + retVal);
    }
else {
    DatagramPacket indp = receiveData(sock);
    processReply(new String(indp.getData()));
    }
sock.close();
}
}
}

```

## D SnmpGet source

```

package smaas.snmp;

import smee.SmartProbe;
import smee.LibraryTable;
import com.adventnet.snmp.beans.*;

import java.net.InetAddress;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.util.*;

import asp.net.AddressNet;
import asp.inet.AddressIP;
import asp.inet.AddressIPv4; // for getLocalHost()

...

public void requireLibraryDependencies() {
    requireLibrary("AdventNetSnmp.jar", search,
        LibraryTable.LIBTYPE_JAR);
    requireLibrary("com.adventnet.snmp.beans.SnmTarget",
        search, LibraryTable.LIBTYPE_JARLIB);

    debug("Requiring libraries");
}

...

/**
 ** Implementation of main probe function
 **
 **/

public void execute(byte[] args) {

    int colonIndex = 0;
    AddressIPv4 destAddr;
    String snmpOut, community, port, destination, oid;
    String stringConvertedArgs = new String(args, 0, args.length);
    SnmpTarget target = new SnmpTarget();
    StringBuffer sb = new StringBuffer();

```

```

debug("starting");

// get the args
StringTokenizer li =
    new StringTokenizer(stringConvertedArgs, ":");

port = li.nextToken();
community = li.nextToken();
destination = li.nextToken();

if(li.countTokens() > 4) {
    report("looking for more than 1 OID");
    report("port: " + port );
    report("community: " + community );
    report("destination: " + destination );
}

try {
    if ( getByName(destination).equals
        (AddressIPv4.getLocalHost() ) ) {
        try {
            // always looking where I landed
            target.setTargetHost("localhost");
            target.setTargetPort(Integer.parseInt(port));
            target.setCommunity(community);

            // get the OID - convert bytes to string
            // set the OID target
            // target.setObjectID("1.1.1.0");
            while(li.hasMoreTokens()) {
                oid = li.nextToken();
                report("oid: " + oid );

                target.setObjectID(oid);

                // do the SNMPGET
                snmpOut = target.snmpGet();

                report("Snmp Output: " + snmpOut);

                sb.append(snmpOut);
                sb.append("\n");

            }
            snmpOut = sb.toString();
            sendReply(snmpOut.getBytes(),
                ((AddressIPv4)probeName.getOriginAddress()));
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
catch (Exception e) {
    e.printStackTrace();
}
debug("exited");
}

```