The ASP EE: An Active Network Execution Environment*

Robert Braden, Bob Lindell, Steven Berson, and Ted Faber
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA

Abstract

This paper describes the ASP Execution Environment (EE), a prototype general-purpose active network execution environment that initiates and controls the execution of Java-based active applications. Features of the ASP EE include support for persistent active applications, fine-grained network I/O control, security, resource protection, and timing services.

1. Introduction

The DARPA active network research program is developing mechanisms for dynamically deploying portable network software in *active nodes*, which may be programmable routers, middle boxes, or end systems. Early active network research used a *capsule* model, in which an *active packet* called a capsule would carry code and data to be executed in each active node (for example [19, 11, 18]). More recent work has used an approach that is often more practical: an active packet carries only a reference to portable code, so each active node that is visited by an active packet can fetch and execute the referenced code [20, 4]. In either case, active networking creates self-deploying network software.

The basic unit of active network programming is called an *active application* or AA. Execution of AA code may affect the data plane, the control plane, or the management plane of a router; active network research projects have experimented with each of these cases. To provide security and portability, AA code typically uses some platform-independent type-safe language, such as Java.

The active network architecture [6] introduces three generic software components to support AA execution: the *execution environment* or EE, the *node operating system* or node OS, and the *user application* or UA. Each AA is written to execute within a specific execution environment. Each EE should be capable of supporting multiple AAs simultaneously. AAs may be highly dynamic but an EE is expected to be a stable feature of an active node. An active node is controlled by a node OS that is expected to support simultaneous execution of multiple EEs.

Finally, an active packet flow is initiated by some user application (UA). A UA might be an application executing within the normal operating environment of a user's end system, or it might be a proxy or other control program within the network, for example.

Two approaches to EE design have emerged, the *strong EE model* and the *weak EE model* [3]. An EE following the strong model is effectively a user-level operating system to control AA execution; AAs obtain most of their essential services, such as execution resources and network I/O, from the EE, while the EE in turn gets its services from the node OS. Under the weak EE model, the EE is only a set of library routines containing useful functions, while AAs obtain their essential services directly from the the node OS.

Under the strong EE model, the node OS must mediate among the EEs and isolate them from each other, and it must protect the node and network by enforcing limits on each of its EEs. An EE must in turn isolate and control AA execution, so that an AA cannot harm other AAs, the EEs, or the node OS. Under the weak EE model, on the other hand, the node OS directly isolates and controls AA execution.

This paper describes a prototype Java-based execution environment, the *ASP EE*.[1] The design of the ASP EE was shaped by the requirements for dynamic deployment of complex control-plane functions such as network signaling and management. The ASP EE therefore includes support for persistent active applications, fine-grained network I/O control, security, resource protection, and timing services. On the other hand, it does not support capsules, i.e., it does

* Funding for the development of the ASP EE was provided by DARPA ITO under contracts DABT63-97-C-0049 (ARP) and DABT63-99-C-0032 (ACTIVATE).

[1] "ASP" stands for *Active Signaling Protocol*, although the ASP EE is applicable to a wide variety of network control applications.

not carry active code "in-band" within an active packet, because a single network-layer datagram is too small to carry code for other than toy algorithms.

Section 2 summarizes the main features of the ASP EE (or simply "ASP"). Section 3 describes the design and implementation of ASP in more detail. Section 4 briefly describes some active applications that have been implemented for ASP, and Section 5 compares ASP with previous work. Section 6 concludes the paper.

2. ASP EE Overview

The ASP EE implements the strong EE model; thus, ASP sub-allocates resources among AAs, isolates AAs from each other, and protects itself and the node OS from the AAs. The ASP EE plays the role of "kernel" to its AAs.

ASP has three major interfaces: "downward" to the node OS, "upward" to its AAs, and (in end nodes) "sideways" to UAs, as indicated in Figure 1.

- **EE/Node OS interface** – The ASP EE was developed to execute under a Unix-based node OS, so its EE/Node OS interface generally follows the Posix standard. An important exception is the network I/O portion of this interface, which is based on the I/O interface of the reference standard [16] for the EE/Node OS interface. This reference interface includes a network I/O abstraction that is based upon *channels* rather than the standard Posix *socket* abstraction. The ASP EE implements this channel abstraction using an adaptation module called *netiod (Network I/O Daemon)*[2, 3]. Netiod executes within Unix and exports an interface based on the channel abstraction.[2]

- **AA/EE interface – PPI** – The AA/EE interface is called the *protocol programming interface* or PPI [17]. The PPI is a "system call" interface that is used by AAs to obtain communication and resource-related services from the ASP EE.

- **UA API** – This interface allows a user application (which may range from a simple GUI to a complex subsystem) to initiate active packets. The UA API is built upon a local IPC mechanism (Section 2.4) to communicate with an instance of the ASP EE on the same node.

The PPI interface includes the following functional areas:

- Process Management (Section 3.1)

[2]For operation within the initial ABone active networks testbed [3], ASP can alternatively be configured to receive input packets from standard input and to send output packets using the JVM-supported socket interface. ASP hides this choice from AAs.

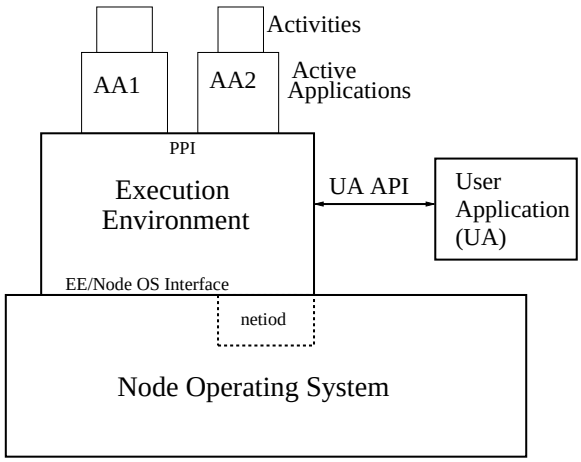

Figure 1. Active Node Architecture

- Network I/O (Section 2.1)
- Timer Services and State Repository (Section 2.6)
- Routing (Section 2.7)
- Traffic Control (Section 2.7)

Like ASP's EE/Node OS interface, the network I/O functions of the PPI are based on the channel abstraction (see Section 3.7). For an EE that used the weak EE model, the EE/Node OS interface would be exposed directly to an AA. Since the ASP EE uses the strong-EE model, it could present different I/O models "upwards" to AAs (i.e., through the PPI) and "downwards" to the node OS. However, the ASP EE uses the channel abstraction model in both interfaces in order to (1) simplify porting an AA to a different EE, and (2) simplify the logic of the EE's implementation of network I/O. We refer to *PPI channels* and *node OS channels* to distinguish the two interfaces. Although both PPI channels and node OS channels use the same channel abstraction, there are important differences in detail that are discussed later (Section 3.8.)

The channel abstraction provides a uniform and flexible interface for demultiplexing incoming packets using packet filters and for sending packets with appropriate encapsulation. It allows an EE or AA to access the native protocol stack of the node OS. The channel abstraction also provides a convenient notation for describing network I/O options.

The ASP EE is designed to support both *persistent* AAs that have persistent execution threads and *transient* AAs that execute once and terminate. For example, a transient AA might be used to query and report local state, while a persistent AA might be used for signaling or routing. Either AA type may maintain persistent state in a node.

A persistent AA may support multiple independent *activities*. For example, a signaling AA should be able to

handle any number of independent signaling activities simultaneously, while a routing AA would typically support a single activity, the local routing computation.

The ASP EE may be configured to load a specific AA at boot time. However, the power of active networking arises from the ability to dynamically load AAs upon request from active packets. Since the code to implement realistic network control algorithms is typically too large for an individual capsule, each active packet for the ASP EE carries an *AAspec* containing a reference to the code for an AA[3].

An AAspec contains (1) an *AAname* that is a globally unique name for the AA, (2) a search path specifying one or more locations from which classes that compose the AA can be fetched, and (3) the name of a primordial Java class specific to the AA and known as the *AAbase* class (Section 3.3). An AAspec is encoded as a variable-length text string in a format defined in the Appendix. A.

As an operating environment for active applications, the ASP EE must provide OS-like functions such as network I/O, resource protection and security, inter-AA communication, and timing services. Resource protection and security are central issues, since an ultimate goal of the ASP EE is to allow safe execution of untrusted user-supplied AAs. To support active networking, ASP needs to support dynamic loading of AA code and to support the UA API. Because the ASP EE is designed for activating complex control-plane functions, it supports two additional features: sharable AA code and dynamic class name binding. All of these features of the ASP EE are discussed in the remainder of this section.

2.1. Network I/O

The node OS and an EE must cooperate to dispatch received active packets to the appropriate AAs and to allow these AAs to send packets into the network. An ASP AA can receive a packet that is addressed to itself or it can intercept a packet from the node's forwarding path. Receiving a packet addressed to the AA requires two demultiplexing steps: (1) the node OS demuxes the packet to the EE, for example using the *typeID* field in the ANEP (Active Network Encapsulation Protocol)[1] header, and (2) the EE demuxes the packet to an AA in an EE-specific manner. If the target AA supports multiple activities, it may have to perform a further AA-specific demuxing to a particular activity.

Intercepting packets makes use of an essential feature of active node architecture [6]: programmable packet filters in the packet forwarding path. An AA can open an *input channel* that will insert a filter to capture an arbitrary subset of the input stream (under security limitations), demuxing packets to the AA via the EE. The ASP EE supports this interception mechanism.

The network control and management applications targeted by the ASP EE require fine-grain control over network I/O. For example, an AA may need to learn the network interface (real or virtual) through which the packet was received or the network-layer source address or the TTL with which a packet arrived. Similarly, an AA may need to explicitly set the source address with which a packet is sent (not necessarily to one of its own interfaces), set the packet TTL, or explicitly control the outgoing interface.

2.1.1 Network Access Modes

The ASP EE supports two network access modes: native IP connectivity and virtual connectivity [3].

- *Native IP connectivity* – This mode gives AAs direct access to the Internet protocol stack, including the ability to intercept packets for which this node is not the destination.

- *Virtual connectivity* – In this mode, ASP EE instances in different active nodes are interconnected by *virtual links* formed by UDP/IP tunnels, creating a virtual topology among nodes running the ASP EE. The IP address and the UDP destination port of a tunnel endpoint define the *virtual link layer (VLL)* address of a *virtual interface*.

An AA operating in virtual connectivity mode may build its own virtual protocol stack on top of the virtual link layer. However, for the convenience of AA builders the ASP EE includes a component called *VNET* that implements a simple user-space virtual protocol stack, as described in Section 3.9. VNET implements virtual layers 2 (link), 3 (network), and 4 (transport) of the OSI protocol reference model. VNET also provides default (unicast) forwarding in the virtual internetwork topology, using a routing table (more accurately called a forwarding information base or "FIB"). ASP EE configuration files define the initial virtual link layer (VLL) interfaces and the initial FIB entries within VNET. An AA with the appropriate permission (Section 2.2) can issue PPI calls to dynamically modify this boot-time information, adding or deleting VLL interfaces or updating routing entries in the VNET FIB. In particular, a privileged routing AA called Jrip (Section 4) updates the VNET FIB to implement dynamic routing in the virtual topology.

2.1.2 Network I/O Primitives

This section discusses the network I/O interface that the PPI presents to AAs. As noted earlier, this interface uses an extended version of the *channel abstraction* of the EE/Node

[3]However, no AAspec is required in the packet when ASP is configured to map particular legacy packets into pre-configured AAspecs, as described in Section 2.3.

OS reference interface [16].[4] ASP implements two kinds of PPI channels: input channels (*InChannels*) and output channels (*OutChannels*).[5] The network I/O implementation in the PPI is described in Section 3.8 below.

The call to open an InChannel specifies a packet filter to select a subset of the input stream. Following the EE/Node OS interface model, the ASP EE uses an *upcall* model to deliver incoming packets to an AA. The *AAbase* primordial object whose name appears in the AAspec must implement the following upcall method[6]:

```
receivePacket(InChannel chan,
    NetBuffer msg,
    Attribute R-attrib)
```

The `chan` parameter specifies the InChannel through which the packet arrived, while the `msg` parameter specifies a receive buffer containing the packet. The `Attribute` object specifies important protocol fields of the packet, including the actual values that matched wildcards in the InChannel parameters (Section 3.7). The `Attribute` object is readily extensible to return additional fine-grained attributes of the packet.

An AA sends a packet with a PPI downcall, using the following method of an `OutChannel` object:

```
sendPacket(NetBuffer msg,
    AddressNet dest,
    Attribute S-attrib)
```

This call specifies the destination address using an `AddressNet` object, which may be an IPv4 or IPv6 address in native IP mode or a VNET network-layer address in virtual connectivity mode. The `Attribute` object specifies other values to be sent in specific protocol fields, overriding OutChannel parameters.

See Section 3 for more information on network I/O in the ASP EE.

2.2. Security and Isolation

The EE protects itself from an untrusted AA by a combination of static and dynamic means. The scope rules of the Java language (e.g., the package mechanism) provide static protection for internal EE variables that need not appear outside the EE. The EE provides dynamic protection of other variables by installing a custom Java Security Manage, which blocks illegal access requests from an AA to either the JDK or the EE.

[4]However, the ASP EE can also be configured to use *socket* I/O to the Posix interface, at the sacrifice of significant flexibility.

[5]ASP does not currently implement the cut-through channels of the EE/Node OS interface spec [16].

[6]The IOException clauses are omitted here from the sendPacket and receivePacket calls.

The AA is the unit of security and isolation within the ASP EE. The ASP EE must prevent *boundary violations* and *resource violations* by an AA. In a boundary violation, one AA makes unauthorized changes to the data or code of another AA or to the rest of the system. In a resource violation, an AA uses to excess some resource that it is permitted to use.

The ASP EE protects against AA boundary violations using Java's strong typing and safe references. Since an AA cannot access a field in a class for which it does not have a reference, one AA cannot access the state of concurrent or previously-executing AAs. The EE is constructed to prevent "leaking" of AA references to other AAs by passing all PPI parameters by value, i.e., as references to cloned copies. This cloning is done by the callee (usually the EE).

An (unprivileged) AA is also prevented by the EE from sending a packet containing an AAspec that is not its own, so one AA cannot spoof another. Finally, each AA is given an isolated subspace of the system file space, and the EE's security manager prevents an (unprivileged) AA from accessing files outside this subspace. File system access is needed at least for utility functions such as reading AA configuration files and writing and manipulating AA log files.

The ASP EE's protection against resource violations is limited to approximate fair-sharing of the CPU among AA threads (see Section 3.1). Other major resources – e.g., memory and network output bandwidth – are not currently protected by the ASP EE.

The ASP EE is intended to support some trusted AAs that perform basic signaling and management functions for the node and which therefore need access to node control functions. However, ASP is also intended to safely execute untrusted AAs. This conflict is resolved by guarding sensitive control functions with permissions that are granted only to privileged AAs; see Table 1. Permissions are granted to AAs by an EE configuration file, which is consulted at the time of AA loading.

There are current limitations of the ASP EE's security mechanism. First, ASP does not support user authentication, so AA permissions cannot be determined by the user or user group that initiated an active packet. There is a proposal for a certificate-based end-to-end authentication system for this purpose [10]. Second, given that AA permissions must be based upon AA identity, ASP should prevent an intruder from spoofing an AAname and causing a rogue version of an AA to load and execute. A code signing mechanism is needed to prevent this.

2.3. Loading AAs

To dynamically load an AA, the EE must receive or intercept an active packet and determine its AAspec. The AAspec may be determined in any of several different ways,

Permission Name	Permission Meaning
interface	Can modify VNET virtual interface table.
route	Can modify VNET route table.
divert	Can perform arbitrary packet intercept.
process	Terminate another AA.
file	Can read/write outside AA file subspace.
socket	Can use JDK sockets interface rather than than channel-based PPI. (Useful for legacy or third party software.)
native	Can load native code into the JVM.
noaaspec	Can suppress EE generation of AAspec in outgoing packets.

Table 1. AA Permissions in the ASP EE

as discussed below. No matter how it is determined, the AAspec contains the AAname and a search path. ASP uses the AAname to locate the AA if it is already loaded, or it uses the search path to locate and load the byte code for the primordial *AAbase* class (Section 3.3) of that AA and instantiate this object. In either case, the EE then passes the packet to the AA via a `receivePacket` upcall in the *AAbase* object. Subsequent references to missing AA classes cause the JVM to use the same search path to load the missing classes.

Loading across the network uses a reliable transport protocol, either the legacy protocol TCP with native IP connectivity or a Java-based implementation of RDP [15] with virtual connectivity. [7]

A search path can include multiple locations; the ASP EE will try each in turn until it succeeds. Each location can be the network address for a code server, either an HTML server or an ASP-specific server (actually a specially-configured copy of the ASP EE.) Alternatively, a location can be an implicit reference to the previous-hop node from which the packet arrived; this option provides hop-by-hop deployment of AA code, a technique introduced by the ANTS EE [20].

The ASP EE implements multiple ways to determine the AAspec for an incoming active packet, to provide flexibility in support of a variety of experimental situations. The AAspec may be: (1) carried explicitly as an ASP header prefixing the AA payload, (2) defined implicitly by an EE-specified packet filter that maps a subset of incoming packets into a pre-configured AAspec, or (3) encapsulated within the AA payload using some AA-specific syntax. Case (3) is intended for interworking with legacy non-active protocols. [8]

The choice among these cases is controlled by an ASP configuration file, `asp.conf`. The three cases are determined by lines of the following forms in the configuration file:

- Case (1) – AAspec explicitly in packet:

```
"InChannel <ChanParms> invokes *"
```

- Case (2) – AAspec implied by ChanParms match:

```
"InChannel <ChanParms>
        invokes AAname <AAname>"
```

- Case (3) – AAspec encapsulated within AA payload:

```
"InChannel <ChanParms>
        invokes Legacy <AAname>"
```

At boot time, the ASP EE will open an InChannel with the parameters specified by `<ChanParms>`, for each line in the configuration file. When an arriving packet (or UA API message; see Section 2.4) matches these parameters and is delivered to the EE, the EE will use the configuration clause following the `invokes` keyword to determine the AAspec. The asterisk in case (1) implies that the AAspec is to be found in the packet; in cases (2) and (3), the configuration line implies an AAname that is mapped into a complete AAspec using another ASP configuration file.

After loading a new AA, ASP will deliver the incoming packet (or UA API message) to that AA via a `receivePacket` upcall that specifies an *implicitly-opened* PPI InChannel. The EE will use the same implicitly-opened PPI InChannel to pass to that AA all subsequent packets that match the same `<ChanParms>` parameters and determine the same AAspec. Such an implicitly-opened PPI InChannel will have the attributes specified in the `<ChanParms>` of the matching configuration file entry. It is possible for an AA to have multiple implicitly-opened InChannels as the result of multiple matching configuration file entries. Once it starts, an AA may explicitly open additional InChannels as well as OutChannels.

The `<ChanParms>` parameter can specify filtering in either the native IP protocol space or in the virtual protocol space created by VNET. The encoding of this parameter is discussed later in Section 3.7.

[7] RDP was built into the ASP EE to provide the experimental option of building an entire active protocol stack, using no legacy protocols above the link layer. See [9] for a description of the properties of RDP and an overview of its implementation in Java.

[8] Note that putting the AAspec into the payload creates an apparent circularity: since the encapsulation is AA-specific, it would seem that the AA must be loaded to scan the packet for the AAspec that is needed to load the AA. This circularity is avoided by mapping the packet to the AAspec for a service-specific transient AA that only extracts the real AAspec from the packet.

2.4. User Application API

The UA API is used by UAs (user applications) to launch activities using specific AAs. The ASP EE uses a TCP connection for IPC between the UA process and the EE process.[9]

UA API TCP connections appear as PPI channels to the AAs. An AA can thus be dynamically loaded by request from a UA as well as by the arrival of an active packet. To initiate an activity, a UA opens a TCP connection to a well-known TCP port in the ASP EE and sends the first request message. The same configuration mechanisms described in Section 2.3 are used to determine an AAspec from the incoming message and to dynamically load the AA if it is not already loaded. The UA's message then arrives at the AA through an implicit UA API channel (Section 3.7). API InChannels and OutChannels are identical; to send a reply to a UA, the AA casts the implicitly-opened InChannel into an OutChannel for a call to `sendPacket()`.

2.5. Inter-AA Communication

The PPI implements an inter-AA communication (IAC) facility, again using PPI channels. An InChannel and an OutChannel belonging to (the same or) different AAs can form a unidirectional IAC pipe. The writer AA creates the pipe by opening an OutChannel specifying IAC, the AAspec of the reader AA, and an IAC pipe name. This creates an implicit InChannel for the reader, which uses a normal `receivePacket()` upcall to receive messages sent by the writer AA. This mechanism assumes that the AAspec and the pipe name will be known to both parties by prior agreement; there is no rendezvous mechanism in the current ASP EE. Pipe data is passed by value, so references cannot leak between AAs. Multiple IAC pipes between the same pair of AAs are distinguished by the connection names set by the writer AA. The reader AA can reject the open request by closing the channel, causing the writer AA to receive an end-of-file exception from its next write request.

2.6. Timer Services

Most non-trivial network control protocols need a timer service for retransmissions, timeouts, sending periodic soft-state refresh messages, etc. An ASP AA can spawn its own persistent (ASP) threads for timing (Section 3.1). However, for the convenience of AAs, the ASP EE provides a timer service combined with a state repository mechanism. The state repository allows an AA to instantiate multiple private name-to-object mappings or "tuple spaces", which are called (soft) *state containers*. Associated with each tuple in a state container are two timers: a *timeout* timer that discards the entry after a specified interval, and a *refresh timer* to periodically trigger an upcall to send refresh messages to neighboring nodes.

A state container is therefore a repository for a set of 4-tuples: `{Key, Value, RefreshT, TimeoutT}`. Here *Key* is a key for retrieving the tuple, *Value* is the object being stored, and *RefreshT* and *TimeoutT* are refresh and timeout time intervals, respectively. Expiration of either timer causes an upcall to a corresponding method specified in a helper class. Expiration of the timeout timer deletes the tuple from the state container before the upcall is performed. If either interval is negative, the corresponding upcall and/or deletion does not occur. A negative timeout time implements hard state, which may be useful for some persistent applications, although soft state is to be generally preferred.

State containers are created and accessed by ASP EE library routines, including methods *get*, *put*, and *remove*. A state container is local to an AA so it cannot allow references to leak to another AA. A state container therefore cannot be used for inter-AA communication; the IAC facility described above must be used instead. An AA must maintain in its local context the handle for each state container it creates.

2.7. Other Services

In order to support control-plane functions, the ASP EE should provide a (privileged) AA with direct control over facilities in the packet forwarding path. This includes access to forwarding tables (FIBs) and packet scheduling mechanisms in both the real and virtual modes. Currently, support in these areas is only partial, and the PPI interfaces are not unified between native IP and virtual connectivity.

In the virtual connectivity mode, the PPI provides the ability to read, update, and receive change notification for the virtual FIB in VNET. There is not packet scheduling currently implemented in VNET.

In the native IP connectivity mode, the PPI provides access to a multicast FIB maintained by mrouted, the multicast routing daemon using the DVMRP protocol. ASP does not currently provide access to the native unicast FIB of the node. An AA does have access to a traffic control interface designed for the RSVP protocol [5] and extended for an active filtering application [22].

[9] Note that a TCP connection works remotely as well as locally, so the UA API can be used to inject an active application into a remote active node, "tunneling" through the Internet. This *remote UA* capability has proven very useful for remote management of ASP EE instances in the ABone [3] active networks testbed.

2.8. Sharable AA Code

The widespread application of active networking technology in the real world may result in the proliferation of large and complex AAs. Furthermore, the ease of deploying new versions of a protocol with active networking may lead to a proliferation of AAs representing different versions of the same service, to introduce new features and for user-specific customization. The result may be many AAs that could share a great deal of code in common.

To reduce the memory footprint for such active applications, the ASP EE supports the sharing of common class byte code among different AAs. The mechanism for accomplishing this in Java is described in Section 3.4. The class inheritance mechanism of an object-oriented language like Java naturally supports such code sharing. Inheritance allows the selective modification of individual methods and fields of a class C_x, by defining a new class version C_y that extends C_x. The common methods and fields of C_x need be loaded only once.

However, the sharable byte code capability creates several technical problems for the ASP EE.

1. Sharable byte code severely restricts the usefulness of the `static` attribute in AAs. Normal Java class definitions may contain data with the `static` attribute, or they may use *static initialization* to set data values when a class is loaded. In either case, data values are created that can be read or written by any AA sharing the same byte code. This would violate the isolation of AAs and hence is not permissible under ASP. The ASP EE must therefore provide a mechanism to replace `static` variables and static initializers in sharable byte code. This replacement is *AA-local data* (*AA-LD*), described in Section 3.2.

2. To make shared code useful, the ASP EE must support *dynamic binding* of class names, as described in the next subsection. Suppose that two AAs share a common class C1 but use different versions of class C2, and that a method M1 in C1 constructs a new instance of C2. The particular version of C2 to be constructed depends upon which particular AA is executing M1. That is, the name used for C2 must be dynamically bound in C1.

3. For reasons described in detail in Section 3.4, the ASP EE implements shared byte code using a single class loader shared among all AAs. This creates a conflict between the ability to unload an individual AA and the ability to share byte code among AAs. In Java, each class loader instance creates a separate name space for the code and data of the classes it loads. Thus, the fully qualified name of a loaded class is implicitly prefixed by a reference to the ClassLoader object that loaded the class. To unload byte code of a class so it can be garbage-collected, it is necessary to delete the ClassLoader that was used to load that class. Individual AA classes cannot be unloaded if there is a single class loader shared among all AAs.

Due to these considerations, the ASP EE can be configured to operate in one of two modes: code-sharing or non-code-sharing.[10] The code sharing mode uses a single class loader and shares common byte code; this mode is expected to be useful for production operations where there is significant overlap of classes among large, complex AAs. In the non-code-sharing mode, each AA has a distinct class loader for each AA and common byte code is replicated in the heap. This mode will be useful when there is significant churn of AA code being installed and removed, e.g., when testing new AAs.

2.9. Dynamic Class Name Binding

Dynamic class name binding can be logically divided into two distinct steps, *name mapping* and *version resolution*.

Name mapping is necessary in AAs for the effective use of shared byte code, as discussed above. In the name-mapping step, an *apparent class name* used as a placeholder is mapped to a *target* class name. This mapping is performed dynamically whenever the AA constructs a new instance of the apparent class, i.e., when it wants to execute the Java constructor `new` using an apparent class name. Name mapping yields the name of the target class to be loaded from a code server and instantiated.

Many operating systems support the dynamic loading of code fragments into running applications from loadable libraries, using a standardized naming convention for representing version and compatibility information. However, Java performs dynamic loading at the granularity of classes rather than libraries, and it has no such naming convention. The ASP EE fills this gap with the optional *version resolution* step of dynamic class name binding. Version resolution maps the target from the name-mapping step, a *versioned name*, to the actual class name. In particular, version resolution can resolve a *wildcard* version specification to find the latest version of the code for a particular class.

The name mapping step of dynamic class name binding is expected to select from among names for functionally different target classes, e.g., classes that implement different feature sets. Name mapping defines the class composition

[10] This simple alternative could be generalized to define sharing groups, with a class loader per group. Each AA would be mapped to a group by its AAspec. Non-shared byte code mode would correspond to every group containing a single AA.

of a particular AA, so this mapping is defined in the primordial *AAbase* class of the AA. On the other hand, the version resolution step is expected to choose among classes that perform identical functions with identical interfaces but may for example represent different generations of debugging and/or optimizing the same class code.

To specify versions, the ASP EE adopts the naming convention:

```
<class name>_<major V #>_<minor V #>
```

Here `<major V #>` and `<minor V #>` represent the major and the minor version numbers, respectively. A new version that did not change the interface or function would have the same major number but a new minor number. The minor version number may be wild-carded ("`*`") to load the latest minor version available from the code server. Note that the major version number overlaps in function with the name mapping step; experience will reveal whether the major version number is redundant with the name-mapping step and could be removed.

The ASP EE uses different implementation strategies for the two steps in dynamic binding. The name mapping step is implemented by explicit AA code (see Section 3.5 below), while the version resolution step is performed by the EE. It would have been possible to perform both name mapping and version resolution entirely inside an AA, allowing different AAs to use different conventions. However, version resolution within an AA may require two round trips to the code server, one to return all available version names and the second to load a specific version. Therefore, version resolution is performed in the ASP EE, which can move wildcard resolution to the code servers.

3. ASP EE Implementation

3.1. Processes and Threads

The ASP EE implements a simple Java-based *process model* to control AA execution. The model includes the definition of a process, a rudimentary process scheduler, and logically separate data spaces for different AAs. The ASP process model is flat, providing no hierarchical structure. There is a one-to-one correspondence between ASP processes and AAs executing in the node.

The ASP process scheduler provides simple round-robin scheduling. This prevents starvation of AA processing, overcoming the undefined semantics of thread scheduling in the Java language.[11] The ASP EE scheduler also measures approximate CPU utilization for each process and the lifetime of the process since inception. This information could be used to demote the priority of processes which are consuming excessive CPU resources or have been in existence for an extended period of time.

Each ASP process may contain an arbitrary number of Java threads that are realized using the `AspThread` class. This class has all the methods of the `java.lang.Thread` class and adds the following two methods:

```
public static AspThread currentThread()
public AspProcess getProcess()
```

The first method returns a reference to the currently-executing thread, and the second returns the ASP process that contains this thread.

To implement InChannels, the ASP EE creates a default AspThread per AA (i.e., per ASP process), to perform `receivePacket` upcalls. The single thread per AA preserves ordering of packets and ensures that there cannot be more than one packet-reception upcall at the same time. The upcall design allows simple transient ASP AAs to be completely event-driven. For the upcall, the default thread is handed to the ASP process using a sequence like the following:

```
AspThread t = new AspThread();
AspProcess p = new AspProcess(...);
...
t.setProcess(p);
<upcall(...)>
t.setProcess(null);
```

The first `setProcess` call hands off the thread `t` to the process `p`, while the second reverts thread ownership back to the EE.

An AA may also fork persistent AspThreads. Most AAs will have at least one such persistent thread, forked internally by the state container library routines to handle their timing. An AA can also do its own timing explicitly using an explicitly-created AspThread.

Java currently disallows the forced termination of an application thread. When an ASP process is terminated, the ASP EE relies on the AA to terminate its outstanding AspThreads. Every AspThread should periodically invoke the PPI call: `boolean isAlive()` and terminate if the result is `false`. Calling the method `terminate()` sends this message to other AspThreads in the same ASP process. The ASP scheduler will demote any threads that are not terminated to minimal scheduling priority.

3.2. AA Local Data

The ASP EE provides a process-local (hence, AA-local) data space to each ASP process. This *AA-local data* or *AA-LD* provides a space for variables that are global within an

[11] The Java specification contains no requirement for preemption or fairness; thread scheduling policies are regarded as an OS implementation detail.

AA but unavailable to other AAs. For example, the state containers implemented by EE library routines use AA-LD rather than static variables for saving context.

The AA-LD mechanism exports two basic methods to an AA:

```
static void putLD(String name,
                  Object value)
static Object getLD(String name)
```

The `putLD` method places an object into the AA-LD data space with the key `name`, and the `getLD` method retrieves the object using the same key. The ASP EE implicitly qualifies the key by the class name of the caller, allowing different classes to use the same key to maintain class-specific data within the AA-LD space. It also enforces Java's static scoping rules on accessing class-specific data.

3.3. The AAbase Class

Every AA includes a primordial Java class, known as the *AAbase* class, that extends the ASP class *asp.AAContext*. To begin loading a new AA, the ASP EE obtains the name of the *AAbase* class from the AAspec, invokes the ASP class loader (Section 3.4) to fetch and load that class, and instantiates it.

The *AAbase* class has the following major functions.

1. It implements upcall routines invoked by the EE (except those associated with state containers). In particular, it must implement the `receivePacket` method.

2. It can be used by the AA for saving local context, as an alternative to AA-LD. This is possible because there is a distinct instance of the *AAbase* object for each AA.

3. It may contain mechanisms for dynamic class name binding: a mapping table and routine, and/or proxy constructor methods. See Section 3.5.

4. It inherits the fields from the corresponding *AAspec*.

A `receivePacket` upcall is executed on the *AAbase* object itself, providing the AA with an immediate reference to the *AAbase* object (using the Java primitive `this`). However, when an AA is dispatched by a timer upcall from a state container, the AA may need a reference to the *AAbase* object to find its local context; this reference can be kept in AA-LD.

3.4. Class Loading

The ASP EE fetches AA classes using search paths found in AAspecs. A reference to a missing class causes the JVM to invoke a loader method of the ASP class loader (that overrides the standard JVM ClassLoader). The upcall from the JVM provides the name of the needed class, and a stack examination by the class loader determines which AA process was executing when the load request occurred.

When the ASP EE is executed in non-code-sharing mode, each AA uses a separate class loader instance and maintains its own copy of all its byte code. Each class loader instance can maintain the corresponding AA search path as member data.

There are several possible design approaches for code-sharing mode. The two most promising are a class loader instance per AA, and a single class loader for all AAs.

- *Class Loader per AA* – Although class loaders create individual name spaces, it is still possible to share common byte code across multiple class loaders, by using a shared table to explicitly share class references between different loader instances. This approach could be used to realize code sharing with one class loader per AA.

  However, the fully qualified name of a class would remain prefixed with the class loader instance that performed the original load operation on that class, and significant complexity would be required to avoid violating the access rules of the Java language. Java symbols that are defined with package scope must be visible to one another. Therefore, an implementation of the class-loader-per-AA approach must not load classes from the same Java package into the name spaces of different class loaders. This requires that the cooperating loader instances delegate the loading of all classes within a given package to a specific class loader instance.

- *One Class Loader* – The ASP EE adopted the approach of a single ASP class loader for all AAs, to avoid the complex inter-class-loader communication required by the class loader per AA approach. The single class loader need only maintain a table containing the search path for each AA, whose identity it can obtain from the currently executing ASP thread.

3.5. Coding Rules for an ASP AA

Active applications running under the ASP EE cannot use completely arbitrary Java code, for reasons of functionality and security that have been discussed earlier. In summary, executing ASP AA code must obey the following rules; see [17] for details.

1. An AA must include a primordial *AAbase* class. See Section 3.3.

2. A persistent application thread must periodically call the `isAlive` function and exit if it returns `false`.

3. An AA may create its own thread(s), but it must not use `java.lang.Thread` or `java.lang.Threadgroup` directly; it must instead use the ASP classes `AspThread` or `AspThreadGroup`, respectively.

4. An AA must use the `AspFile` class instead of the `File` class for accessing the local file system.

5. An AA must use no Java `static` fields; it must instead use AA-local data as described in Section 2.8.

6. An AA must use no Java static initializer blocks; it must instead use a replacement construct [17] that causes a static initializer to be invoked once for each new ASP process that is created. The ASP implementation executes only the initializers that are reachable by a given process; this is determined by a static reachability analysis on the loaded byte code, augmented by dynamic reachability information gained by monitoring the usage of getName(). This mechanism requires that an AA use `AspSystem.getName()` instead of `Class.getName()` and use `AspSystem.forName()` instead of `Class.forName()`.

7. An AA must use no synchronized static methods; it must instead place locks on global variables defined in its AA-LD space.

8. When a new instance of an AA class is constructed, name mapping should be provided wherever it may possibly be useful for future flexibility.

Restrictions (3) and (4) and (6) concern particular Java library classes that must be replaced by ASP versions. This substitution of class names can be performed automatically on the byte code by the ASP class loader. This is currently implemented for cases (3) and (6) above, and it could easily handle all such substitutions. Restrictions (5) - (8) are not required if an AA is to be executed only in ASP EEs executing in non-shared-code mode.

Finally, we summarize the coding necessary for the name mapping part of dynamic class name binding. The primordial *AAbase* class contains the AA's code and data for name mapping. The procedure for this mapping is in principle AA-specific, but an example is given here. In the *AAbase* class, one could build a `java.util.Hashtable` object for name mapping:

```
Hashtable nameTbl= new Hashtable();
nameMapper.put(<apparentName>,
                          <targetName>);
   ...
```

When it needs to construct an instance of a mapped name, the AA might call a method in the *AAbase* object that uses this hash table to map a string `nameArg` containing the `<apparentName>` into a target name:

```
String targetName =
      (String)nameMapper.get(nameArg);
```

and then invoke the ASP EE routine:

```
AspSystem.forName(targetName).
```

The *forName* routine would resolve a versioned name if necessary, fetch the resulting actual class, and load the class code. The AA would finally use the Java reflection interface to construct an instance of the actual class.

The detailed AA coding for this dynamic name binding is somewhat complex, especially when the constructor is parametrized. If version resolution is not required, class name mapping can be much simplified by the use of *proxy constructor* methods [17] in the *AAbase* class to implicitly do the name-mapping step. Instead of directly invoking the Java `new` primitive to create an instance of a class, an AA would call a corresponding proxy constructor method in the *AAbase* class. There would be a proxy constructor method for each `<apparentName>` in use. For example, one proxy constructor method might be:

```
<superclass>   new<apparentName> {
    return  new  <targetName>();
}
```

Here `<superclass>` is a parent of `<apparentName>` and `<targetName>`.

3.6. Network Interfaces

The once-simple concept of a network interface has become complex, due to multiple network-layer addresses per physical interface, virtual interfaces for multicasting, IP-in-IP tunnels, and support for both IPv4 and IPv6, for example. As a result, a node OS may know about many more logical/virtual network interfaces than physical network interface devices.

The ASP EE builds a single master interface table that contains all logical, virtual, and physical interfaces known to the node OS as well as the virtual interfaces created by VNET. This table, built at boot-time from kernel queries and configuration information, may be modified by subsequent PPI calls that add or delete network interfaces.

The *logical interface number* or LIN is the ordinal position of an entry in this master interface table, so every defined interface has a unique LIN value. When an interface is deleted its LIN is not reused; thus, the space of LIN values, initially contiguous, may become quite sparse over time.[12]

[12] This design has the theoretical problem that it prevents the ASP EE from running forever.

protocolSpec	Receive Attributes (addressSpec)	demux Key?
vif<LIN>/vn/vt[/asp]	local_Vport, remote_Vport, local_Vaddr, remote_Vaddr, tos, ttl, LIN	
vif<LIN>/vn/rdp [/asp]	local_Vport, remote_Vport, remote_Vaddr, LIN	
vif		Yes
if<LIN>/ipv4/udp[/asp]	local_port, remote_port, local_addr, remote_addr, LIN	
if<LIN>/ipv6/udp[/asp]	local_port , remote_port, local_addr, remote_addr, LIN	
if<LIN>/ipv4/udp/rdp[/asp]	local_port remote_port, remote_addr, LIN	
if<LIN>/ipv4/tcp[/asp]	local_port remote_port, local_addr, remote_addr, LIN	
if<LIN>/ipv6/tcp[/asp]	local_port remote_port, local_addr, remote_addr, LIN	
if<LIN>/ipv4[/asp]	local_addr, remote_addr, protocol, TTL	Yes
if<LIN>/ipv6[/asp]	local_addr, remote_addr, protocol, TTL	Yes
api[/asp]	local_port remote_port, local_addr, remote_addr, LIN	(n/a)
ipc	dst_AAspec, chan_name	(n/a)

ProtocolSpec Elements:			
Symbol	Meaning	Symbol	Meaning
vif<LIN>	Virtual link layer (VLL) processing in VNET.	if<LIN>	Link layer processing.
vn	Network-layer datagram processing by VNET (LayerVN or Layer VNS).	ipv4 ipv6	Network-layer IPv4 datagram processing. Network-layer IPv6 datagram processing.
vt	Transport-layer processing by VNET (LayerVT or Layer VTS).	udp	Transport-layer UDP datagram processing.
rdp	Transport-layer RDP processing.	tcp asp	Transport-layer TCP processing. Process ASP header (framed AAspec).

Table 2. Supported PPI Channel Parameters

PPI calls allow an AA to find a particular interface object in the master table and to get a copy of the complete table [17].

The receive attributes set by a `receivePacket()` call include the LIN number for the network interface on which the packet arrived, and an AA can set the LIN value in the send attributes parameter to the `sendPacket()` call. However, some existing kernels may not provide complete information about, or control over, its real network interfaces.

3.7. The Channel Abstraction

The EE/Node OS reference interface [16] specifies the syntax and semantics for three basic parameters of the network I/O channel abstraction – protocolSpec, addressSpec, and demuxKey [16, 3]. These three parameters define packet processing and filtering to be performed on incoming active packets by an InChannel, while the protocolSpec and addressSpec alone are used to specify headers to be added by an OutChannel.

- *protocolSpec* – This parameter is a character string that defines a set of protocols to process and remove headers from an incoming packet, or to be added to output packets.
- *addressSpec* – This parameter defines specific values for demultiplexing fields in the protocol headers listed in the protocolSpec. For a PPI channel, the ASP EE encodes the addressSpec as an *attribute* object containing binary values, rather than as the character string used in the EE/Node OS interface specification.
- *demuxKey* – This parameter specifies arbitrary filtering on the payload of input packets selected by the addressSpec after processing by the protocols named in the protocolSpec. The demultiplex key may contain one or more (offset, length, mask, value, relop) tuples.

For example, the protocolSpec `"if4/ipv4/udp"` for an InChannel implies that a packet received on logical interface number 4 will be processed by IPv4 and by UDP; the UDP payload may be further filtered by a demuxKey that selects which UDP payloads will be delivered.

The `<ChanParms>` element in InChannel entries of the `asp.conf` configuration file (Section 2.3) is actually composed of strings representing protocolSpec, addressSpec, and an optional demuxKey; see [17] for the detailed formatting.

3.8. PPI Channels

Table 2 summarizes the protocolSpecs currently supported by PPI channels. It also shows the corresponding receive attributes, i.e., the attributes included in a `receivePacket()` upcall, and whether demuxKeys are supported. A subset of these attribute fields is available for opening PPI channels and for a sendPacket call; see [17] for full details. An attribute list generally plays the role of an addressSpec; however, ASP attributes include some protocol fields that may not be included in the EE/Node OS spec definitions of an addressSpec but may be of importance to an AA.

Table 2 also illustrates the extensions to the node OS channel abstraction that are supported by PPI channels.

- The ASP EE generalizes the channel abstraction to cover the virtual protocol stack implemented by VNET as well as the native IP stack. Thus, the protocolSpec `"vif13/vn/vt"` implies a packet received on the virtual interface with `LIN = 13` and processed by the virtual network layer (`vn`) and the virtual datagram protocol (`vt`); see 3.9.
- PPI channels are used to implement the UA API (protocolSpec `api`); see Section 2.4.
- PPI channels are used for inter-AA communication (protocolSpec `ipc`); see Section 2.5.
- PPI channels support stream-based I/O (protocolSpec elements `rdp, tcp`) although the channel abstraction is fundamentally datagram-based. In particular, ASP supports the reliable transport protocol RDP [15] in addition to TCP.
- The element `asp` appended to the protocolSpec requests that the EE process explicit AAspec headers. Here "process" means to remove the AAspec header from the AA payload in an InChannel and to automatically supply an AAspec header in an OutChannel. In fact, an unprivileged AA is *required* to append `asp` to its protocolSpecs, or the channel open request will fail. This mechanism (mentioned in Section 2.2) prevents an unprivileged AA from sending a packet with an AAspec not its own.

Stream-based channels (TCP, RDP) have some special rules. They do not preserve packet boundaries from sender to receiver. A connection is opened actively or passively by opening an OutChannel or an InChannel, respectively. Once open, the stream-based channel is actually duplex: an InChannel object can be cast to an OutChannel object and vice versa. A stream channel provides an *end-of-file* upcall to the AA if the other side closes the connection.

3.9. VNET: Virtual Protocol Stack

The ASP EE includes a software package called VNET [14], which implements a simple virtual stack that may be

useful to AAs. VNET supports only unicast transmission on (virtual) point-to-point links – no broadcasting, multicasting, or multiaccess. It does not support fragmentation or reassembly in the virtual internetwork layer. Its network-layer address structure is flat, providing only host addresses with no network numbers. On the other hand, VNET does include a *hop-by-hop delivery* mechanism, which provides a service analogous to the use of the IP Router Alert option [12]. A datagram sent using hop-by-hop service is addressed to a final destination but is delivered to the next hop as if it had been addressed there. Packets sent without the hop-by-hop service option are said to receive *end-to-end* service.

VNET provides a Network Management Interface (NMI) to the PPI, allowing a (privileged) AA to create, modify, or query the network interface and FIB tables used by the VNET protocols.[13] An AA can also register upcall routines that will inform the AA that something in these tables has been changed.

VNET currently implements the following virtual protocol layers.

- *LayerVT, Layer VTS – user datagram transport*. LayerVT provides a datagram transport service analogous to UDP, using source and destination ports chosen from a VNET port space of 32-bit integers. Incoming datagrams are queued internally for delivery to AAs and dropped if these queues are filled. LayerVTS is the same but provides hop-by-hop service.

- *LayerRDP – Reliable byte stream*. This layer provides an interface to a Java implementation of the Reliable Data Protocol (RDP) [15]. VNET does the buffering and protocol conversion to make a sequence of RDP packets look like a byte stream, i.e., like a TCP connection.

- *LayerVN, LayerVNS – connectionless network*. These layers provide a connectionless network service analogous to IP but with fewer features (see above). LayerVN provides end-to-end service while LayerVNS provides hop-by-hop service. LayerVN packets carry a time-to-live (TTL) field to handle routing loops; LayerVNS needs no TTL.

- *LayerUI – Virtual Link Layer using UDP/IP*.

  LayerUI packets are framed using a HeaderUI header and encapsulated within UDP and IP headers. A local AddressUI object is supplied as the virtual link layer address when a LayerUI object is created. The LayerUI implementation opens a socket for the local IP address and UDP port number contained in the AddressUI object. A listening thread receives datagrams from this socket, decapsulates the packet, and sends it to the appropriate Layer 3 protocol for subsequent processing. Packets passed down from Layer 3 are framed with a HeaderUI header and sent to the appropriate UDP socket for encapsulation.

- *LayerAnep – Virtual Link Layer using ANEP/UDP/IP*.

  LayerAnep frames packets using both a UDP/IP header and an ANEP header but no VNET-specific header. Virtual link layer addresses for this layer use AddressAnep objects that contain an IP address, UDP port number, and the ANEP Type ID.[14]

[13] This NMI should be extended to control native IP connectivity.

3.10. VNET Implementation

The current VNET implementation uses a general object-oriented framework to implement its protocols. A particular VNET protocol layer is defined by three Java classes: `address`, `header`, and `layer`. The `layer` class implements the layer-specific protocol processing rules, an `address` object names an instance of an entity at that layer, and the `header` class defines the representation of the protocol header in a packet. The layer names in the previous section are in fact `layer` class names.

To process a packet, each VNET `layer` object maps the appropriate local address – source or destination for sending or receiving, respectively – for its own protocol layer to the next `layer` object instance with that address, and hands the packet to that `layer` instance for subsequent processing. The VNET protocol stack within a given ASP EE instantiation will have `layer` instances corresponding to all addresses currently in use at that node. This will generally include network-layer interface addresses, the corresponding link layer addresses for these interfaces, and transport layer addresses in use by running applications. Only one layer instance can be bound to a particular local VNET port at any time.

Packets that arrive carrying non-local addresses, such as packets that are destined for the forwarding engine, cannot be mapped to specifically-named layer instances. To handle this case, each layer designates a `layer` instance as the "default" to handle all packet requests for unknown address values.

4. ASP AAs

The features of the ASP EE can be illustrated by describing a sample of ASP AAs that have been demonstrated. See Table 3.

[14] LayerAnep uses UDP port zero as an escape to support the reception of VNET packets on standard input, as required by the initial ABone management tool Anetd [3].

Name	Function
Jrip	RIP routing protocol
PIM	Protocol-Independent Multicast.
Jrsvp	RSVP QoS signaling protocol
AFSP	Active Filter Signaling Protocol
Delay	Generate random forwarding delays
Sencomm	Active network management
	environment (BB&N).
(many)	EE monitoring and management

Table 3. Some ASP-Based Active Applications

- *Jrip* – The Jrip AA executes the RIP protocol in the virtual topology defined by the VNET configuration file. Using a privileged PPI call (permission `route`, see Table 1), Jrip updates the FIB that is used by VNET for default forwarding in the virtual network layer topology.

- *PIM* – This AA, which does multicast routing and forwarding using the PIM sparse-mode protocol, was originally written for the ANTS EE but ported to the ASP EE as an exercise. Porting from ANTS to ASP is much easier than the other direction, primarily because ANTS does not support persistent AAs or timing services.

- *Jrsvp* – This very large and complex AA implements most of the QoS signaling protocol RSVP [5], for both unicast and multicast data traffic. It handles signaling traffic and data traffic using either virtual or native IP connectivity.

- *AFSP* – This signaling protocol for interest filtering [22] for large-scale distributed simulations has been demonstrated. The AFSP protocol is quite close to RSVP, so the AFSP AA was derived from Jrsvp by changing a small percentage of the Java code.

- *Delay* – This AA is indicative of the usefulness of active networking for quickly generating and deploying experimental tools. It was developed by the ACC (Active Congestion Control) research project [8], which has been using the ASP EE as an experimental vehicle.

- *Sencomm* – This complex AA is itself an execution environment, recursively built upon the ASP EE. It was developed by BBN as an active network monitoring environment.

5. Comparison to Other Work

Pioneering research in active networks preceded the development of the standard active network architectural model [6], with its clear distinction between EE and AA. These early projects explored important ideas, for example, data flow architectures (Netscript [21, 7]) and a secure scripting languages (PLAN [11]). The earliest EE conforming to the AA/EE/Node OS architecture was the ANTS system[20]. ANTS is the project whose objectives and general architecture are most nearly akin to those of the ASP EE. Both follow the active networks architecture, executing on top of a node OS to provide a well-defined environment for executing active applications. ASP has borrowed useful ideas from ANTS. Each has its strengths and weaknesses; neither is the final answer for an active networks EE.

ANTS uses an elegant mechanism to propagate and execute transient AAs hop-by-hop. Although its design was inspired by the capsule model, ANTS "capsules" actually carry AA code by reference. This mechanism was the inspiration for the corresponding ASP facility, specifying the *PHOP* (previous hop) option in the search path of an ASP AAspec. The ANTS EE has seen considerable use as a platform for experimentation with AAs and node OSs. One of its important features is a strong resource control and security mechanism, including a generalized TTL to limit how many packets a particular AA can originate. One significant drawback of ANTS is its lack of persistent AA execution threads, and therefore its inability to provide timer services within the network. One research project using ANTS added persistent AAs, at the sacrifice of ANTS' generalized TTL control on packet transmission [13].

In several important aspects the ASP EE represents a more realistic model for real-world active network technology than ANTS. The ASP EE supports persistent AAs, including threads, timer services, etc., which ANTS does not. Both EEs support virtual connectivity, but the ASP EE also supports native IP connectivity, which is perhaps more important. ASP provides a more general AA-launching mechanism mechanism than the ANTS `Application` class. The ASP EE has paid attention to providing programmable router services like access to native-mode routing tables and fine-grained network I/O.

On the other hand, while the ASP EE represents a significant step beyond ANTS in its facilities, some useful features of the ANTS EE are missing from ASP. The University of Utah has modified ANTS to support the full EE/Node OS interface, while ASP supports only a generic version of the network I/O portion of that interface. This makes ANTS more easily portable to specialized node OSs, although it is unclear that this will be a real issue. ASP has no mechanism comparable to ANTS' limit on the use of network resources, although we do not believe that the ANTS mech-

anism has sufficient generality for real control plane functions. Finally, there are important gaps in the ASP security system; in particular, ASP would benefit by adopting a code signature mechanism like ANTS, to ensure that an AAname cannot be spoofed.

6. Conclusions

The ASP EE was developed to explore of the use of active networks for rapid deployment of complex protocols in the control and management planes. This general objective led to dynamic code installation, functional extension and customization of code, and the use of portable code to reduce protocol standardization. The ASP EE was designed to provide these facilities with fine-grain, flexible, and portable access to network protocol layers, using the channel abstraction. A second primary objective of ASP was the isolation and resource protection to allow arbitrary AAs to safely execute, while privileged AAs can still access key node resources. Other important ASP EE features include support for persistent AAs, timer services and a soft-state data repository, a general user application model, and an inter-AA communication mechanism.

Much of the complexity of ASP results from a desire to provide a highly flexible platform for active application experiments. The generality of ASP partly reflects the evolving orientation of the overall active networks research program while ASP was being developed. For example, the implementation of the virtual protocol stack within the VNET component of ASP resulted from an early push from the program sponsors towards "reinventing" networking using active networking. The program orientation later shifted towards the use of active networking within the context of legacy Internet protocols, which makes the VNET virtual protocol stack and forwarding in the virtual topology less useful. On the other hand, active routers are very likely to be interspersed with non-active routers, so some support for overlays is likely to continue in importance.

Certain features of ASP, for example sharable byte code, turned out to introduce much greater complexity than expected. Although byte code sharing may be desirable in a production environment, it may not be worthwhile in the current active network experimental environment. However, the ASP EE can be configured to bypass byte code sharing.

As the preceding discussion has made clear, there are important directions for further work on the ASP EE. These include:

- Per-AA limits on the use of the JVM memory heap.
- Support in OutChannels for packet scheduling to enforce limits on AA network usage, at least fair-sharing.
- End-to-end user authentication to control AA permissions.
- Signed AA code.
- A complete implementation of PPI channels (e.g., filling many of the blank spaces in Table 2).
- A complete implementation of access to native IP routing tables.

Finally, we list larger design issues raised by the ASP EE.

1. The ASP EE is (too) specialized for support of persistent AAs. This is revealed in the fact that ASP can support at most one execution instance of a given AA at one time. Supporting multiple simultaneous executions would require the introduction of a new identifier space. This space might provide an "AAidentifier" or "AA flow id", which could also be used for efficient demuxing of packets from the nodeOS directly to an AA, even in the strong EE model. This concept might be pushed to an "activity identifier", to incorporate activities in a clean and efficient manner.
2. An active packet for ASP can invoke at most one AA, which raises the question of how to compose multiple AAs to form a single logical AA. This is an open research issue.
3. The ASP EE does not support capsules. However, this either/or approach to capsules may be too simple. In the future, capsules may prove useful in a specific and limited fashion to make small incremental changes, e.g., for customization, of large and complex AAs that are initially loaded by reference.
4. Java is powerful, well-documented, and widely available, but it may not be the ideal language for active network programming.

7. Acknowledgments

Jeff Kann, Graham Phillips, and Craig Milo Rogers made significant contributions to the design, implementation, and testing of the ASP EE and its AAs. Additional contributions to the ASP effort were made by Alberto Cerpa, Hugh Choi, Yu He, Siva Jayaraman, Vivek Shenoy, Sarjana Sheth, and Ya Xu.

We are also grateful to those in other organizations who have used the ASP EE and given us bug fixes and feed back. We particularly acknowledge Alden Jackson and Regina Rosales of BBN, and Steve Zabele and Mark Keaton of TASC.

A. AAspec Format

An AAspec consists of a set of variable-length quoted text strings, denoted by "Q-" prefixes in the following BNF-like syntax definition.

```
<AAspec>  ::= <Q-AAname>
        [<sp> <Q-searchPath>
         <sp> <Q-ClassName-List> ]

<Q-AAname>     ::= " <AAname> "

<Q-searchPath> ::= " <SearchPath> "
<SearchPath>   ::= <location> |
          <location> , <SearchPath>
<location>     ::= <URL> | <ASPloc>

<ASPloc> ::=
   asp-private-ip://<IPaddr> |
   asp-private-vnet://<VNETAddr> |
   asp-private-ip://PHOP

<Q-ClassName-List> ::=
               " <ClassName-List> "
<ClassName-List>   ::= <className> |
   <className> <sp> <ClassName-List>

<AAname>   ::= <AAnameChar> |
               <AAnameChar> <AAname>
<AAnameChar> ::= <letter> | <digit> |
    ! | # | $ | % | & | = | + | - |
    _ | @

<className ::=
        <Fully-qualified Java class>
<IPaddr>   ::=
        <host name or numeric IP addr>
<VNETAddr> ::= <32-bit integer>
<sp>     ::=  < 1 or more space chars>
```

`AAname` is assumed to be globally unique.

`<SearchPath>` contains a list of code-servers from which the EE will fetch the AA classes. A code server may be an HTTP server (`<URL>`) or an instance of the ASP EE running as a code server (`<ASPloc>`). In the latter case, the byte code can be fetched using a TCP connection (`asp-private-ip`), using RDP through VNET (`asp-private-vnet`), or from the previous hop using TCP (`PHOP`) over native IP.

The `<ClassName-List>` entry is a list of classes to be loaded initially. The first (and normally the only) entry on this list will be the name of the *AAbase* class.

The optional `<Q-searchPath>` and `<Q-ClassName-List>` elements may be omitted only if it is certain that the AA has already been loaded, e.g., because it is loaded at boot time by EE configuration.

The following string is an example of an AAspec. The newlines included here are for presentation purposes only, as AAspecs contain no newlines.

```
"RSVP_base_version"
"asp-private-ip://PHOP,
    asp-private-ip://128.9.160.128,
"rsvp.VersionBase"
```

An AAspec appearing explicitly as the header of a packet must be framed with a 2-byte length field that precedes the variable-length AAspec string. This convention is also used for messages across the UA API.

References

[1] D. S. Alexander, B. Braden, C. A. Gunter, A. W. Jackson, A. D. Keromytis, G. J. Minden, and D. Wetherall. Active network encapsulation protocol (ANEP). Draft RFC, `ftp://www.cis.upenn.edu/pub/switchware/ANEP`, 1999.

[2] S. Berson, R. Braden, and E. Gradman. The network I/O daemon - netiod. Technical report, USC/Information Science Institute, October 2001. `http://www.isi.edu/abone/techspecs.html`.

[3] S. Berson, S. Dawson, and B. Braden. Evolution of an active networks testbed. Submitted to DANCE, 2002.

[4] R. Braden. Active signaling: the arp project. *In OPENSIG Workshop, University of Toronto*, Toronto, CA, October 1998.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP): Version 1 functional specification. RFC 2205, Sept. 1997.

[6] K. Calvert, Ed. Architectural framework for active networks, version 1.0. July 1999. `http://www.cc.gatech.edu/projects/canes/publications.html`.

[7] S. daSilva, D. Florisi, and Y. Yemeni. Composing active services in netscript. *Position paper, DARPA Active Network Workshop*, Tucson, AZ, March 1998.

[8] T. Faber. Acc: Active congestion control. *IEEE Networks*, May/June 1998.

[9] T. Faber. *An Implementation of the Reliable Data Protocol for Active Networking*, 2000. `http://www.isi.edu/active-signal/ACC`.

[10] T. Faber, B. Braden, B. Lindell, S. Berson, K. Bhaskar, and S. Schwab. Active network security for the ABone. 2002. `http://www.isi.edu/abone/techspecs.html`.

[11] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunger, , and S. M. Nettles. Plan: A packet language for active networks. *In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, 1998.

[12] R. Katz. IP router alert option. RFC 2213, Feb. 1997.

[13] D. Kiwior and S. Zabele. Active resource allocation in active networks. *IEEE Journal on Selected Areas in Communications*, March 2001. Special Issue on Active and Programmable Networks.

[14] B. Lindell. *The ASP Virtual Networking (VNET) Component*, 2000. `http://www.isi.edu/active-signal/ARP/docs.html`.

[15] C. Partridge and R. M. Hinden. Version 2 of the Reliable Data Protocol (RDP). RFC 1151, Apr. 1990.

[16] L. Peterson, Ed. Node OS interface specification. July 1999. `http://www.princeton.edu/nsg/papers/nodeos99.ps`.

[17] G. Phillips, B. Braden, J. Kann, and B. Lindell. *Writing an Active Application for the ASP Execution Environment – Release 1.2*, 2000. `http://www.isi.edu/active-signal/ARP`.

[18] B. Schwartz, W. Zhou, A. W. Jackson, and et. al. Smart packets for active networks. *Technical Report, BBN Technologies*, January 1998.

[19] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. In *http://www.tns.lcs.mit.edu/publications/ccr96.html*, 1996.

[20] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. ANTS: Network services without the red tape. *IEEE Computer*, April 1999.

[21] Y. Yemini and S. da Silva. Towards programmable networks. *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, October 1996.

[22] S. Zabele, B. Braden, M. Keaton, and B. Lindell. Active multicast information dissemination. *In preparation*, 2002.