ASP EE: An Active Execution Environment for Network Control Protocols

Bob Braden Alberto Cerpa Ted Faber Bob Lindell
Graham Phillips Jeff Kann

USC/Information Science Institute
{braden,cerpa,faber,lindell,graham,kann}@isi.edu

December 3, 1999

1 Introduction

The ARP (Active Reservation Protocol) project at ISI is developing a framework for implementing and deploying complex network control functions using an active network approach. A major goal is to allow dynamic installation of new and modified services as *active applications* (AAs) written in platform-independent code. This approach could significantly reduce the need for protocol standardization, since a (single) implementation of a protocol need interoperate only with itself. In effect, the code defines a private protocol among the instances of the AA running on different nodes. The ARP project is concentrating on AAs that implement network signaling protocols such as RSVP, although the techniques being developed should be more widely applicable.

This portable code approach could be applied in a variety of language and system environments, but the ARP project is currently using Java. Java provides nearly-portable code as well as dynamic linking and loading. This document therefore describes the ARP approach using object-oriented terminology. Thus, we speak of the basic building blocks for code and data as classes, which can be instantiated as objects.

1.1 Active architecture

Research in active networks has resulted in a hierarchical model for organizing the software within an active node, illustrated in Figure 1. The ARP project is building upon this basic design.

An AA executes within some particular *Execution Environment* or EE. For example, an EE might include a Java virtual machine (JVM) and a set of control and interface libraries. The node is under

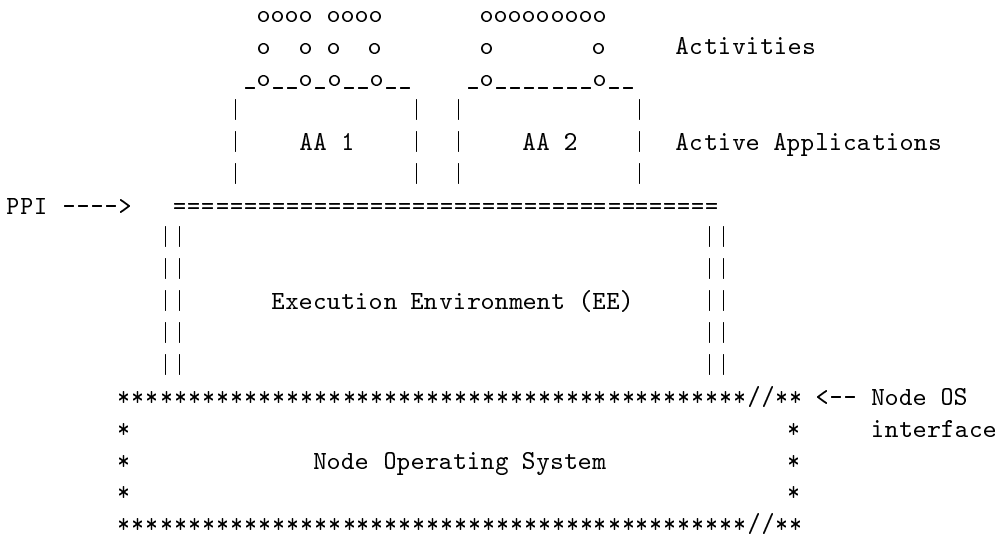


Figure 1: Active Architecture

control of a node OS, which must be able to support multiple EEs. The node OS must mediate among EEs and isolate them from each other. Different EEs may have different trust levels, but to the extent that they are not fully trusted, the Node OS must enforce both boundary and resource limits on each EE.

Each EE is capable of supporting multiple simultaneous AAs. The EE is effectively an operating system for AA execution, with a well-defined (and protected) interface to EE services that we call the *protocol programming interface* or PPI.

An AA may be capable of handling multiple independent signaling or control *activities*, as shown in Figure 1. For example, the fundamental unit of signaling activity for an RSVP AA is an RSVP session, and a given RSVP AA may be handling any number of sessions independently and simultaneously. On the other hand, a routing AA may be able to support only a single activity. Each AA will have its own rules for resource sharing, multiplexing, and isolation among different activities.

An active packet that arrives at a router must be delivered into the appropriate activity. At least in principle, the demultiplexing takes place in steps.

1. The Node OS demultiplexes the packet to the appropriate EE.

   If the packet uses ANEP [1] encapsulation, then the ANEP header specifies the EE. However, an EE must set a packet filter to intercept *legacy* protocol packets that do not contain an ANEP header.

2. The EE then determines which AA should process the packet.

   There may be an EE-specific header defining the AA; if not, the EE must have some way to extract the AA specification from the AA-specific packet. The format of this information

may be service-specific, and the process of extracting it is complicated by dynamic loading of AA code. Alternatively, packet filters might be used to map packets to AAs. For example, particular CIDR classes of source and/or destination addresses might be used for AA selection.

3. The AA must determine the activity to which the packet belongs.

   This mapping is completely AA-specific.

1.2 EE Overview

As an operating environment for active applications, an EE must be able to provide the following general functions.

- Dynamic loading of remote AA code.

  A fundamental requirement of active networking is that AA code be dynamically loadable over the network. AA code can be carried within protocol data packets, which are then called *capsules*, or it can loaded out-of-band from the protocol data.

- Security and Resource Protection.

  The EE must prevent boundary and resource violations between AAs and between an AA and the EE. It should be possible for AAs to collaborate, but accidental or malicious interactions must be prevented. Note that AAs will typically be less trusted than EEs; ultimately, it may be possible to execute user-supplied (and hence completely untrusted) AAs within an EE.

  Once an AA has been injected into the network, it should always be possible to identify the source of the injected AA, because without such identification, it would be difficult to track errant AAs.

  Security and protection are discussed further below.

- Controlled inter-AA communication.

  Although isolation of AAs from each other is the default, there should be a mechanism to allow AAs to deliberately share information.

- Network I/O.

  The node OS and the EE together must be able to dispatch received active packets to the appropriate AAs and to send packets into the network.

- UA API

  There must be some way to initiate active services. We refer to the initiator as a *user application* or UA.

  If the active packets are providing an end-to-end service, the UA will execute in an end system. The UA might also be a proxy or other control program to invoke an active service that is internal to the network. For "permanent" internal active services such as routing protocols, a system configuration program may play the role of an AA to start the AA at boot time.

- Timing Service

  Any non-trivial network control protocol will need a timer service, to control retransmissions and/or soft-state timeouts, for example.

In more detail, an EE needs to support the following security-related features.

- EE Protection.

  The EE must also be able to protect itself from AAs. In addition, there must be no avenue that allows a malicious AA to circumvent the EE's mediation between applications and resources.

- Application Isolation.

  We distinguish two aspects of isolation: protection against boundary violations and protection against resource violations. A boundary violation means that 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 some resource that it is permitted to use, but uses it to excess so that it interferes with the operation of another AA or the system. The EE must isolate an AA to prevent either type of violation.

  In practice, it is difficult to completely eliminate the side effects of shared resource allocations. For example, some applications may be affected by a different application that exhausts some shared system resource. The EE must mediate resource utilization – processor, memory, disk, and network – in an attempt to minimize any inter-application interference.

- Prevention of application spoofing.

  Network managers may require that active packets maintain an invariant source address (or invariant identifier) so that misbehaving AAs can be detected. One way to achieve this property is for the EE protect certain header fields in the packet from being modified by any AA. Particular fields that may be sensitive, in the case of the ASP EE, are the source address and the AAspec (discussed below).

- Inter-AA Communication.

  Cooperating AAs should be able to communicate and exchange desired information. Some form of EE supported IPC should be provided to applications.

- Auditing.

  Auditing of security sensitive actions should be configurable, based on policy.

2 The ASP EE

The ARP project is developing a Java-based execution environment called the ASP EE. Here ASP stands for *active signaling protocol*, even though the ASP EE is designed for a wide variety of network control applications, not just signaling.

The ASP EE necessarily supports the general EE features presented earlier: dynamic loading of AA code, security and resource protection, network I/O, a UA API, controlled inter-AA communication, and a timer service. The ASP EE mechanisms for these features are presented in later sections of this document.

Because the ASP EE is designed for activating complex network control protocols, it supports the following features which are especially suited to that task.

- Out-of-band dynamic loading of AA code (no capsule support),
- Sharable AA code,
- AA data isolation,
- Fine control over network I/O,
- Dynamic class binding, and
- Soft-state containers.

These choices are discussed in the remainder of this section.

This document considers only AAs running under the ASP EE. There may be other AAs simultaneously running under other EEs on the same node with ASP AAs. In the remainder of this document, "the AA" refers to an ASP AA, and "the EE" refers to the ASP EE.

2.1 Out-of-Band Dynamic Loading

In principle, each protocol message could be an active *capsule* containing the algorithms to be used to process that packet. In practice, real network control algorithms are typically too large to be carried in individual signaling packets, which can only carry names and locations for fetching the AA code out-of-band. The ASP EE must be able to dynamically fetch AA classes from remote code servers and load them into memory on demand. AA classes can be removed from memory when they are no longer needed, i.e., when there are no active execution contexts for the AA. However, for efficiency it will be useful to cache AA code in memory or on disk.

When an ASP packet arrives, the EE demultiplexes it to the appropriate AA, after loading that AA if it was not already present. This demand loading and demultiplexing is controlled by a packet data field that is called an *AAspec*. The AAspec may encapsulate the rest of the AA-specific payload, or it may be embedded within that payload.

An AAspec has two functions: (1) to demultiplex incoming packets to a particular AA and (2) to provide the receiving EE with a path to servers that contain the implementations of the classes that compose the AA. An AAspec consists of a variable-length text string containing three pieces of information:

- a globally unique name for the AA,
- the name of a primordial AA class known as the *AAbase* (discussed later).
- a search path specifying one or more code servers from which classes that compose the AA can be fetched.

The current format of an AAspec is defined in Appendix B.3.

In principle, only the first packet received for a given AA need specify the AAbase class name and search path; the AAname alone is sufficient for demultiplexing incoming packets once the AA has been loaded. However, many AAs will have to deal with asynchronous arrival of protocol packets, so that it will not be possible to be certain which packets will arrive first. Furthermore, the AA code, once loaded, is only cached; it might timeout and be replaced in memory during a pause in an AAs activities. For simplicity and robustness, therefore, we require that every protocol packet carry a complete AAspec.

(If an AA is loaded at boot time its complete AAspec is known and packets might carry only the AAname.)

2.2 Sharable AA Code

While an active node is expected to support at most a few different EEs, the ASP EE must be able to support a large number of different AAs. We expect that active networking will allow the introduction of a significant number of new control and management services into network nodes. For the more complex services, there will be a proliferation of different versions of the same service, each implemented as a distinct AA.

Protocol updates often introduce new features or correct errors in earlier versions of the protocol. AAs may also implement the same protocol differently depending on its resources; an AA that is primarily, but not exclusively, a server may use different data structures than an AA that is primarily a client.

Many standard protocols have optional features, which lead to much of the complexity in many real life protocols and in the programs that implement them. The protocol standardization process tries to find the simplest useful feature subset, but the result is generally an engineering compromise that cannot satisfy all user needs. Protocols are evolving entities, and new features are added.

In summary, an AA implementing a new version of a service may be introduced into network nodes for any of the following reasons.

- A network manager might install an AA version that changes only the implementation, but not the protocol, to fix bugs or improve processing performance.

- A network manager might introduce an AA version with a new feature set in the protocol, for all users of that service.
- A network manager might introduce an AA with a new feature set in the protocol, for a specific subset of the users of the AA.
- A particular user or group of users of the AA might introduce an AA that implements a protocol variant with their own private feature set.

AAs for real signaling protocols such as RSVP are large and complex, but different AAs for the same service will often share a great deal of common code. To significantly reduce the memory footprint for active applications, the ASP EE supports the sharing of all class byte code among different AAs. To prevent unwanted interaction between AAs, the AA byte code must be read-only pure procedures. Pure procedures are procedures that do have global side effects.

The class inheritance mechanism of Java naturally supports sharing of code. Inheritance allows the selective modification of individual methods and fields of a class Cx, by defining a new class version Cy that extends Cx. The methods and fields of Cx are available in Cy, but their byte code need be loaded only once to have instances of both Cx and Cy active at the same time.

However, sharable code introduces several technical problems.

- Java class definitions may contain embedded data with the `static` attribute, or they may use *static initialization*; either of these conflict with pure procedures, because the data is accessible to or the side effect can be detected across all instances of those classes *in all AAs.*

  In order to provide code sharing but still isolate different AAs, classes making up an AA must be pure procedures. However, static data has an important function: it provides a generalized form of global data that can be accessed from multiple classes. In particular, it is needed to provide an *anchor* for locating all object references local to an AA.

  To resolve this conflict, the ASP EE provides a new mechanism to support AA-local data (*AA-LD*); AAs must use this mechanism in place of static variables and static initializers. The AA-LD mechanism is described in Section 4.2.

- To make shared code useful for network control protocols, the ASP EE must support dynamic binding of class names. Suppose that a method M of a shared class constructs a new instance of some other class. The particular name of the class to be constructed depends upon the AA that is executing M. This situation is likely to arise when two AAs implementing different versions of the same service share common class code. For example, the class created by M could provide encryption services that must be implemented one way for use inside the United States and one way for export.

  To handle this case, the constructed class name needs to be *dynamically bound.* Section 2.5 describes how such dynamic class name binding is achieved with the ASP EE.

- Sharable class code must be implementable using the standard Java class loader mechanism and the partitioning of access that it creates. The resulting design issues are discussed in section 3.3.

2.3 Data Isolation

Although class code is to be shared among AAs, one AA should be unable to access data of another AA, except by explicit agreement. The AA is the natural unit of data isolation in ASP.

The basic isolation mechanism in the ASP EE uses Java's strong typing and safe pointer variables. An AA cannot access a field in a class for which it does not have a reference. The EE is constructed to prevent references "leaking" from one AA to another. Since AA class code is read-only, references cannot leak through the code. Measures are taken at the AA/EE boundary, i.e., the PPI, to prevent AA internal references leaking through the EE. This mechanism is described later in Section 5.

We use the term *local context* for the complete, self-contained set of values and mesh of references maintained by a single AA. Much of this local context is typically organized into tables and list structures, which the AA must consult in processing arriving packets and timer events. There will generally be a set of *anchors* for these data structures that the AA uses to locate the rest of the local data.

2.4 Fine Control over Network I/O

The ARP project is concerned with network control AAs, which need complete network-layer control over sending packets and access to all the information in incoming packets. In fact, ASP needs to know more about an incoming packet than its full contents, for example, ASP uses information about which interface received a given packet. Other information that ASP or another network control EE may need includes:

- Knowledge of which network interface a packet arrived on, and control of which interface a packet is sent out.
- Knowledge of the network-layer source address with which a packet arrived, and control over the source address with which a packet is sent.
- Knowledge of the network-layer hop count (e.g., IP TTL) with with a packet arrived and control over the hop count with which a packet is sent.

The concept of a network interface has become complex and subtle, due to multiple network-layer addresses per physical interface, virtual interfaces for multicast and other types of tunnels, and support for multiple network layer protocols (e.g., IPv4 and IPv6). This complexity is reflected in the ASP EE as a variety of (network) interface objects. The EE creates these objects, generally as the result of a configuration or discovery process at boot time. The AAs have read-only access to them.

The ASP EE uses different methods to sending and for receiving packets.

- An AA receives packets via upcall from the EE. The EE invokes the following method in the *AAbase* object:

```
int     receivePacket(InterfaceNetIO inf, NetBuffer msg,
                      AttribRcv attrib)
```

  The `msg` parameter specifies a datagram. The `inf` parameter specifies the network (or API) interface through which the packet arrived. Finally, the `attrib` parameter conveys an extensible set of fine-grained attributes of the packet, such as source address and hop count.

- An AA sends packets using a downcall to the EE. The AA invokes the following method of the `InterfaceNetIO` object:

```
int     send(NetBuffer msg, AddressNet to, AttribSnd attrib)
```

  The arguments to `send` perform analogous functions.

This design was chosen to allow ASP to be completely event-driven.

Another aspect of the network I/O model of the ASP EE is the need for flexible interception and demultiplexing of input packets.

- An AA needs to be able to process transit packets that are not addressed to the node but rather are being forwarded by the (router) node. Thus, there must be some packet filter in the input stream to intercept some particular subset of packets. There are also two choices in such interceptions: the intercepted packet might or might not be forwarded as well as passed upward to an AA.
- It is necessary to handle various "legacy" situations where the standard ASP demultiplexing information is unavailable but can be deduced for particular subsets of the input stream.

To handle these problems, the ASP EE uses the *Nchannel* (Network channel) abstraction. An Nchannel a logical path for input packets from some data source to a particular AA or with a particular algorithm to determine the AA. Nchannels are configured in the EE with filter parameters that specify some subset of the incoming packet stream and whether to intercept promiscuously or greedily, and they deliver all such intercepted packets to particular AAs.

An Nchannel might deliver packets to a particular named AA. Alternatively, it might specify the name of an AA that knows how to scan the packet sufficiently to extract an embedded *AAspec*. Finally, it might specify no particular AA, but instead obtain the *AAspec* from an ASP header.

For example, an AA that implements some version of RSVP might take input from an Nchannel that specifies IP protocol number 46 packets, or even protocol 46 and RSVP version 7.

In addition to their hard-wired demultiplexing function, Nchannels connect various link and network layer sources to the EE's standard upcall mechanism for passing packets to an AA. For this purpose, Nchannel classes are runnable, and each Nchannel uses one or more Java threads.

There are different Nchannel types for different types of sources. Nchannel types currently include:

- An IP protocol 46 intercept using a native-mode kernel interface for legacy RSVP,
- the UA API, and
- datagram I/O to the VNET package.

The API Nchannel is also used for output. In the absence of a channel, fully-demuxuable packets can still be delivered to AAs. (Should there be a default or null channel for consistency ??)

Note that Nchannels are orthogonal to interfaces; the channel through which a packet arrives is independent of the network interface through which the packet arrived. Nchannels are internal constructions within the EE; interfaces are known to the AA, as explained earlier.

The Nchannel abstraction is generally related to the input channel abstraction being suggested for a node OS.

(Explain how channel abstraction subsumes UA API)

2.5 Dynamic Class Name Binding

In dynamic class name binding, an *apparent name* used in AA code may be a place-holder that is dynamically replaced by the *actual name* of a class whose code is to be loaded from a server. This replacement occurs when the AA constructs a new instance of the apparent class, i.e., when it executes `new` ⟨apparent class name⟩.

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

1. Name Mapping

   First, a selection is made from among functionally different target classes, which for example may implement different protocol feature sets. We refer to this selection as name mapping. The name mappings used by a particular AA are fundamental to the definition of that AA, and therefore the mapping information is logically defined in the AA's *AAbase* class.

2. Version Resolution

   The target of name mapping will be a class with a particular set of functions, but this class may still exist in multiple *versions*. For example, these versions may represent different generations of debugging and/or optimizing the same class code.

   In many operating systems, dynamic loaders can load code fragments in running applications. These code fragments reside in dynamically loadable libraries, and these libraries adopt a well-knon naming convention for representing version and compatibility information. In Java,

dynamic loading occurs at the granularity of classes rather than libraries, and there is no naming convention for representing versioning or compatibility information. The dynamic class name mapping adds version resolution support to the ASP EE, in particular supporting dynamic resolution of a *wildcard* version specification to find the "latest" version of the code[1].

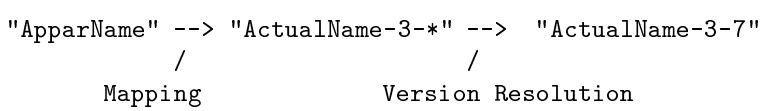
Dynamic class name binding is therefore considered to first map an apparent name into a *version name*, and then to resolve the version name into an actual class name.

To specify versions, the ASP EE adopts the standard naming convention used by many operating systems. Each class can optionally be named using the convention:

```
<class name>-<major version number>-<minor version number>
```

Two classes that share both the same apparent class name and the same major version number are assumed to be both interface- and functionally- compatible. The minor version number is stepped for a new implementation that does not change the interface or functional compatiblity from previous versions, and whose major version number is therefore unchanged. The minor version may be wild-carded (`"*"`) to locate and load the lastest minor version available from the code servers[2].

For example, the two-step replacement process for an apparent class name `ApparName` might be:



In theory, both name mapping and version resolution could be performed entirely inside the AA, allowing different AA programmers to use different conventions. However, technical considerations led to a decision to use the EE to perform version resolution. Resolution of a wildcard version specification within the AA will require two round trips to the code server; 1) to return all available version names to the AA; and 2) to load the specific version picked by the application. It may prove preferable to do this resolution at the code server. It is therefore desirable to have a single convention for version specification (shown above), and to enforce this convention by doing the version resolution in the EE, perhaps in cooperation with code servers.

Thse ideas are captured in the following process that is used in the ASP EE for dynamic class name binding.

---

[1] In object-oriented terminology there are actually two forms of versioning, class versioning and instance versioning. Instance versioning, supported in many object-oriented databases, provides the capability to maintain distinct versions of the same data as the contents of a database evolves over time. Class versioning, on the other hand, corresponds to schema versioning in database terminology. Instance versions are not currently supported in the ASP EE. In this document, an unqualified "version" refers to class versioning.

[2] Experience will reveal whether the major version number is actually redundant with the mapping step and could be removed.

- The primordial *AAbase* class contains the AA's map for dynamic mapping of apparent names to definite names. binding map. The procedure for this mapping is in principle AA-specific, but the EE supplies a library routine `nameMapper` for the purpose.
- `nameMapper` assumes that the *AAbase* contains a table with entries of the logical form:

  ```
  <type> <apparent name> = <version name>
  ```

  Here $< type >$ is the class name from which the definite class must be derived or an interface that it must implement. The $< type >$ information allows the library routine to confirm that the definite class implements the required methods.
- The AA then passes the version name to the ASP EE routine `versionLoad` to resolve any wildcard minor version suffix, fetch the resulting actual class, and load the class code. See Appendix A.3 for examples.
- The Java reflection interface is then used to construct an instance of the actual class. This is somewhat complex if there are constructor parameters (see Appendix A). However, Appendix A also suggests a more efficient approach, in which the AAbase class contains *proxy constructors* embodying the name mapping rules. The proxy constructor approach is especially useful if there is no wildcard version resolution.

2.5.1 Components

We introduce the notion of a *Component*. A *Component* provides a hierarchical name space in which to resolve apparent names to definite names. It can be used to provide a logical grouping of interrelated classes that form a functional unit. Internally, the component maintains a mapping table between apparent names and the definite names, or more generally versions, for the set of cooperating classes.

Dynamically binding the names of all classes constructed within an AA provides considerable flexibility for constructing new versions of the service, but all class names need not be dynamically bound. There is some overhead associated with dynamic binding, so the selection of classes to be dynamically bound trades efficiency against flexibility for code-reuse.

2.6 State Containers

The ASP EE has a state repository mechanism that allows each AA to maintain multiple private name-to-object mappings, which we call tuple spaces. The mechanism supports both soft-state and hard-state. The containers also provide a timer service that invokes upcalls to the protocol code at appropriate times.

Soft state is state that is discarded if it is not periodically refreshed. In contrast, hard-state must be explicitly removed. The soft-state approach has two advantages, simplicity and robustness.

Managing soft-state is simpler because explicit *modify* and *remove* operations are unnecessary, and the same operation is used to *create* and to *refresh* state. The greater robustness comes because hardware and software failures cannot result in dangling state, and because the repeated refresh messages can adapt to changes such as new network routes.

Each AA can instantiate multiple tuple spaces known as *state containers*. A state container is a repository for a set of 4-tuples:

(*Key*, *Value*, *RefreshT*, *TimeoutT*).

Here *Key* is a key for retrieving the tuple, and *Value* is the object being stored. *RefreshT* and *TimeoutT* are time intervals in milliseconds. Expiration of either RefreshT or TimeoutT makes a upcall to a method specified in a helper class. Expiration of TimeoutT deletes the tuple from the state container before the upcall is performed. Either interval can be negative, in which case the corresponding upcall and/or deletion does not occur; this can be used to store hard-state.

State containers are created and accessed by ASP EE library routines. Available access methods include *get*, *put*, and *remove*. Since a state container is local to an AA, a state container cannot cause references to leak to another AA. The AA must maintain the handle for each state container it creates in its local context.

3 Design of the ASP EE

The preceding section explained some of the key design decisions in the ASP EE. This section explores the EE design incorporating these decisions. A discussion of the process model is contained in Section 4.

3.1 Processes and Threads

The ASP EE implements a simple Java-based process model to control execution of multiple AAs. The model provides the definition of process, a rudimentary process scheduler, and logically separate data spaces for different AAs. The ASP EE implements each AA with a process, so there is a one-to-one correspondence between processes and AAs in the node.

All data associated with a given process is completely isolated from other processes using the strong typing mechanism implemented in the JVM. Each process is provided a process-local (hence, AA-local) data space AA-LD for global variables.

A single process may have zero or more persistent Java threads. These must be a modified thread known as an ASP thread (Section 4). The ASP EE is designed to permit purely event-driven AAs, which receive control via upcalls when a packet arrives or a timer goes off. These upcalls are executed

on ASP threads that originate in the EE and are transient; their lifetime is the duration of the upcall. An AA may also fork its own (ASP) threads, which persist until they are killed. Most AAs will have at least one such persistent thread, forked by the machinery of the state container library routines to handle timing.

For a more detailed discussion of the ASP process model, see Section 4.

3.2 AA Local Context

For every AA, there is a primordial class known as the *AAbase* class. This class must extend the EE's *AAcontext* class.

The EE is requested to load an AA either at boot time as the result of configuration or upon arrival of a protocol packet containing an *AAspec* that names an AA that is not already loaded. This *AAspec* also normally contains the class name of the *AAbase* class and a path to a server where classes for this AA are located. The EE invokes the ASP class loader (Section 3.3) to fetch and load the *AAbase* class, and instantiates it. A reference to this object is saved in the process context.

The *AAbase* class has three major functions.

1. It implements the upcall routines invoked by the EE (except those associated with state containers). In particular, the *AAbase* class implements the `receivePacket` method described in Section 2.4.
2. It can be used by the AA for saving anchor references into its local context, as an alternative to AA-local data.
3. It contains the mapping table for dynamic class name binding (Section 2.5) or proxy constructors (Appendix A.3).

The *AAbase* class also inherits an object that contains the fields from the AA's *AAspec*.

Each time it is dispatched, an AA will typically want to have a reference to the *AAbase*, in order to find its local context. The AA can keep this reference in AA-LD; it may also be convenient to keep the reference within its own data structures. A `receivePacket` upcall is executed on the *AAbase* instance itself, so its reference is immediate (`this`). A timer event upcall may, however, need to find the *AAbase* reference from AA-LD or by having the reference embedded within the data object (??).

3.3 Class Loaders

The ASP EE fetches AA classes from code servers using a search path that may have arrived in an *AAspec* carried by a protocol packet. It might also be established by a configuration file at boot time.

During the execution of an AA, references to additional classes can cause the immediate dynamic loading of code. These new classes should be loaded using the search path originally supplied at the start of execution of the AA.

Class loaders in Java are responsible for loading code into the JVM. Each class loader, by definition, forms a separate name space for the code and data that is loaded by that class loader. This allows two different class loaders to load unique copies of the same class, even though they share the same fully qualified name. Essentially, each fully qualified name is implicitly prefixed by the (instance) name of the class loader that loaded the class. Code or data can be shared across class loaders if an implementation of the class loader explicitly shares class references between different loader instances by means of a global table. In this case, the fully qualified name remains prefixed with the class loader instance that performed the original load operation.

Class loading requests originate from the JVM and invoke a load method of the appropriate class loader. This upcall provides the name of the needed class but does not provide the referring class which initiated this load request. Examination of the current thread of execution from within the class loader does provide implicit information about which application was executing when the load request occurred. This information can be used to map a load request back to the originating AA.

Class loading strategies and AA search paths become entwined in the following discussion as we attempt to arrive at a design that loads AA code, shares common code among different AAs, and applies the appropriate search path to each individual class loading request.

If the ASP EE had adopted the Java applet model, each AA would use a separate instance of a class loader and maintain its own separate copy of all class code. An instance of a class loader for a particular AA would then use the search path originally provided in the *AAbase* throughout the execution lifetime of the application. Since there is a one-to-one mapping between AA and class loader instances, each instance of a class loader could maintain a search path as member data and use this search path for all loading requests. This solution is simple and its implementation would be very straightforward, but would discourage code reuse.

ASP did not use the applet model but instead chose a solution which allows the sharing of class code among applications. With this model there does not need to be a one-to-one mapping between AAs and class loader instances. There are three possible alternative design approaches for obtaining the desired sharing functionality, as we now discuss.

- Class Loader per AA

  One approach would be to maintain a class loader instance per AA, analogous to the Java Applet model, but to modify the class loader so that ASP classes, unlike applets, can be shared between AAs rather than being loaded multiple times into separate class loader name spaces. This sharing can be accomplished by making each class loader instance share code with other AAs by populating a shared table that is accessible to the different instances of class loaders. As explained above, a shared class retains the loader namespace prefix of the class loader that originally loaded that code into the JVM. This class loader namespace scoping interacts with the package accessibility rules defined in the Java language. Normally, the symbols from two different classes are accessible to each other if they are in the same package and the symbols

are qualified with the package-scope access modifiers. If two different classes from the same package are loaded by separate class loaders, symbols that are defined with package scope are not visible to one another because they were loaded into separate name spaces. To avoid violation of the package accessibility rules defined in the Java language, an implementation of code sharing by multiple class loaders must therefore be careful not to load classes from the same package into the name space of different class loaders. An implementation of one class loader per AA and shared code requires that the individual loader instances cooperate in such a fashion as to delegate the loading of all classes within a given package to a specific class loader instance.

- Class Loader per Package

  Another approach to shared code, using a class loader per package, would avoid the problems just raised. However, it could lead to an excessive number of class loader instances with no obvious additional benefits. In addition, with this approach there would no longer a one-to-one correspondence between applications and class loaders, so an additional mechanism would be needed to match a load request to the proper search path.

- One Class Loader

  A third possible approach to shared code uses a single ASP class loader for all AAs. This avoids the complex inter class loader communication required for the first two solutions. However, it still requires a mechanism in the class loader to match a load request to the proper AA search path. One solution is to build a table of search paths provided by AAs inside the class loader. When a load request was performed, the class loader can then map the current thread of execution back to a given AA, and then obtain the AA's search path from the table.

The ASP EE has chosen to implement the last (single class loader) method. It provides the necessary functionality required for our EE and avoids the complexity of implementing cooperation necessary to properly obey package scoping rules.

The single ASP class loader loads arbitrary application byte code subject to certain restrictions. These restrictions are as follows:

1. An application can not load or directly reference `java.lang.Thread` or `java.lang.Threadgroup`. Applications are required to be written using the class `AspThread` and `AspThreadGroup` (see Section 4).
2. No static class data is allowed. Applications are required to use AA-local data in their place. Examples are provided in a subsequent section.
3. No static initializer blocks are allowed. A new construct for static initializer blocks is defined in this implementation as a replacement.
4. No synchronized static methods are allowed. Locks cannot be placed on shared class code between processes. Applications can synchronize by placing locks on global variables defined in their AA-local data space.

5. An implementation should avoid using `Class.forName()` and instead use `AspSystem.forName()`. This is necessary to ensure the correctness of the static initializer block execution.

Currently, only restriction (1) is enforced by our implementation of the class loader.

4 ASP Process Model

The ASP EE implements a simple Java-based process model to control execution of multiple AAs. The model provides the definition of process, a rudimentary process scheduler, and logically separate data spaces for different AAs. It is implemented completely in Java to enhance its portability to different JVM implementations.

The ASP EE implements each AA with an ASP process, so there is a one-to-one correspondence between processes and AAs within the ASP EE. Data associated with a given process is isolated from other processes using the strong typing mechanism implemented in the JVM. Each process is provided a process-local (hence, AA-local) data space AA-LD for global variables.

A process is defined by subclassing the AspProcess class. The process model is flat, providing no hierarchical structure at this time. Each process is considered to be a schedulable entity and may contain an arbitrary number of threads.

4.1 Scheduler

The process scheduler provides simple round-robin scheduling. This scheduler overcomes the undefined semantics in the Java language with respect to thread scheduling. According to the Java language specification, it is an implementation detail how threads at the same priority are scheduled; there is no requirement that a preemptive strategy be implemented, to guarantee that all threads at the same priority are given a chance to run. In addition to preventing starvation, our scheduler keeps track of approximate CPU utilization for each process and the total lifetime of the process since inception. This information could be used to demote the priority of processes which are consuming too much CPU resources or have been in existence for an extended period of time. These types of penalties have not been implemented in the current scheduler.

Each application should be able to run in almost complete isolation of other applications. We use the term *almost* because it is difficult to completely eliminate the side efforts of shared resource allocations. For example, some applications may be affected by a different application which exhausts the space within a filesystem. An execution environment will be required to mediate resource utilization in an attempt to minimize any inter-application interference.

4.2 AA Local Data

The AA-LD mechanism exports two 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 repository with the key `name`, and the `getLD` method retrieves the object using the same key. The key is implicitly qualified 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 prevents a class from accessing the data from another class in violation of the access modifiers specified by this other class for the functions that access class-specific data.

4.3 AspThreads

Threads are created using the `AspThread` class. This class has the same methods at the `java.lang.Thread` class, with the following three exceptions:

- `AspThread` has no `setPriority()` method.

  Applications are not allowed to change the priority of threads, since thread priority is now under the control of the ASP process scheduler.

- `AspThread` adds the following two methods

  ```
  public static AspThread currentThread()

  public AspProcess getProcess()
  ```

  The first method returns the currently executing thread. This method is similar to the function in `java.lang.Thread` except that the return type is an `AspThread` rather than a `Thread`.

  The second method return the ASP process that contains this thread.

`AspThreads` may be created by the ASP EE and subsequently handed off to applications to execute upcalls. Here is an example of how this handoff is coded.

```
AspThread t = new AspThread();
AspProcess p = new AspProcess(...);
...
t.setProcess(p);       // handoff thread to application
upcall(...)
t.setProcess(null);        // revert thread back to kernel
```

While threads may be created in a common thread group, this practice is discouraged inside the EE for threads that will be used to upcall into applications. Two threads, which are part of the same Java `ThreadGroup` and handed off to two separate applications, will create an avenue of unintended sharing via the `ThreadGroup` data structure. In the above example, the `AspThread` is created with no specified `ThreadGroup`. The internal implementation of the ASP process model will create a distinct `ThreadGroup` for this thread thus allowing this thread to be handed off to an application in isolation to the threads used by other applications.

4.4 Using the ASP Process Model

The following examples show how to transform Java application code to run under the ASP Process model. Primarily, it is necessary to replace the usage of static data, block initializers, and locks by the use of AA-local data.

1. Replacing static variables (fields).

   In the following example, we are going to transform the usage of a private static variable, `str`, in class bar to a set of functions by the same name that use AA-local data instead of a static variable. The access modifiers on this pair of functions should always match the original access modifier used on the static variable. In this example, we used the access modifier `private`.

   The key used to specify the AA-local data, in this example `str`, is qualified by the class name of the caller. This allows methods in different classes to use the same key to maintain class specific data within the AA-local data space. It also prevents a class from accessing the data from another class in violation of the access modifiers specified by this other class for the functions which access class specific data.

   The following code:

   ```
   public class bar {
       private static String str;
   }
   ```

   should be replaced by:

   ```
   public class bar {
       // Access to "str"
       private static String str() {
           return((String) AspProcess.get("str"));
       }
       // Assignment to "str"
       private static String str(String x) {
           return((String) AspProcess.put("str",x));
       }
   }
   ```

2. Replacing static initializers.

   Static initializers will be called once for each new process that is created. The implementation executes only the initializers which are reachable by a given process. This is accomplished by performing a static reachability analysis on the byte codes loaded by the class loader augmented by dynamic reachability information gleaned from a processes usage of the `Class.getName()` functionality during it's execution. Given that the EE cannot directly monitor the usage of `Class.getName()` it is required that applications use the function `AspSystem.getName()` as a replacement.

   A static initializer is distinguished by the following prototype:

   ```
   public static void name(AspStaticInitializer)
   ```

   Programmers are free to name the static initializers as they choose.

   The following code:

   ```
   public class bar {
       static {
           str = new String("test");
       }
   }
   ```

   should be replaced by:

   ```
   public class bar {
       public static void anyNameYouWant(AspStaticInitializer x) {
           AspProcess.put("str",new String("test"));
       }
   }
   ```

3. Replacing static locks.

   The following code:

   ```
   public class bar {
       private static synchronized void sync() {
          ...
       }
   }
   ```

   should be replaced by:

   ```
   public class bar {
       private static void sync() {
           Object lock = AspProcess.get("lock");
           synchronized (lock) {
            ...
           }
       }
   }
   ```

5 Security and Isolation

Each unique *AAspec* defines a separate application activity that is isolated from other applications. One application cannot access the state of a second application, whether the second application is running simultaneously with the first, or whether the second application left behind some state in a previous epoch. In this section, we describe how the ASP EE isolates itself from applications and isolates applications from each other.

5.1 Static-scoped isolation

The package mechanism of Java naming in conjunction with Java access rules provides only coarse-grained isolation. The coarse-grained isolation may not be sufficiently general enough for the EE because the EE may want to provide difference access restrictions to different applications. Package-scoped isolation is limited because is cannot prevent two application activities from accessing some object based on the *identity* of the activity. In future, we envisage that fine grained access to the EE will be controlled by a security manager.

5.2 Dynamic-scoped isolation

In the Java applet model, isolation between applets is achieved by instantiating a separate class loader for each applet, which then occupies an independent name space. However, the ASP EE allows the controlled sharing of code within a single address space by using a single class loader instance to load all applications.

Because all applications run in a single address space, a single application must be able to remain isolated by preventing references to its own internal structures from leaking to other applications. In addition, references that an application passes to the EE across the PPI must not be leaked to other applications.

To understand leaking references via the EE, consider the following sequence of events. The EE returns a reference to an internal `yy` object to the application. Then the application assigns a field `xx` in object `yy` to some other object. Finally, the EE returns the same internal `yy` object to a second application. The result is that the field `xx` in the `yy` object is now shared by both applications, which violates isolation.

There are two possible forms of isolation:

- *Read/Write Isolation* means that there is no reference in one application that is also a reference in a second application. That is, the accessible memory spaces of both applications are disjoint.
- *Read-Only Isolation* means that there is no reference in one application that is writable by a second application.

The ASP EE enforces *Read/Write Isolation*.

To further the discussion, we introduce the following two terms. An *in* parameter is a reference that is passed into a method and an *out* parameter is a reference that is passed out of a method.

The *Read/Write Isolation* requirement is achieved if one of the following two conditions hold:

- The *out* parameter is also an *in* parameter and the callee (e.g. the EE) does not keep a reference of the parameter.
- The *out* parameter is passed by value.

Here *by value* means a reference to a separate copy of the parameter object. Furthermore, any references embedded within that object parameter is similarly copied, and this copy process is applied recursively. In Java terminology, we pass a reference to object a *clone* of the original object. Note that cloning is done by the callee (e.g. the EE) and is transparent to the caller.

Ensuring that all *in* parameters are passed by value is insufficient by itself, because one cannot prevent the leaking scenario described above. In practice, however, it is generally wise to pass *in* parameters by value also, to prevent unexpected and perhaps obscure side effects. Again, the cloning is done by the callee and is transparent to the caller.

One further requirement regarding the cloning of parameters is that the caller should not be able to override the `clone` method of a parameter, because if the caller could, the callee cannot guarantee that the object was passed by value. To prevent the caller from overriding the `clone` method, the classes of all parameters are scoped `final`.

For security reasons, the AAspec typically cannot be modified by AAs (an exception is discussed in ??). In other words, users are prevented from specifying the location of their classes. This might be done by having a local configuration file that ensures that classes are loaded only from explicitly listed locations. Particular users would only be allowed to specify their own locations if they were explicitly listed in the configuration file.

5.3 Access Rules

(Explain how Java access rules should be used, including any gotchas for security/isolation)

A Writing AAs for the ASP EE

A.1 Coding Conventions

AA code to use the ASP EE must obey the following rules.

- AA classes cannot use static fields; such data must be stored as an (attribute, value) pair as AA-local data (AA-LD) or in the *AAbase* object.
- Static methods should generally be avoided, to provide maximum opportunity for dynamic binding.
- When constructing a new instance of an AA class, dynamic class name binding should be included wherever it is may possibly be useful for future flexibility. See Appendix A.3 for coding examples.

Note that all of these rules imply some performance penalty.

We can generally characterize the classes composing the AA as either *code-only* or *state-containing*. State-containing classes are used to represent state blocks and other variable state, which are typically instantiated many times; these are part of the local context. Code-only classes, which basically contain only code (i.e., methods), would not require instantiation at all if method overloading were not required. However, dynamic class name binding is only possible for instance (i.e., non-static) methods, and therefore one instance of each code-only class must be instantiated. References to such code-only objects are also part of the anchor information that must be stored as AA-local data (AA-LD) or in the *AAbase* object.

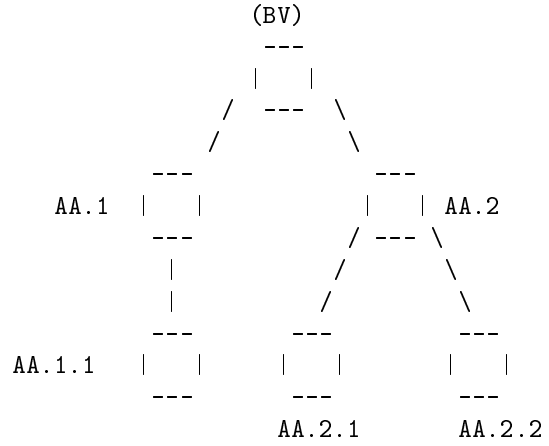
A.2 Derivation of AAs

There is no formal requirement for a relationship among different AAs versions for a given network control service. The ASP EE framework allows each AA to be created, independently of others, by just defining its *AAbase* object. However, in most cases new versions of an AA will not be developed in a complete vacuum; more commonly, a new AA will be in large part derived from an earlier AA for the same service.

It is useful as an organizing principle to visualize this derivation relationship among different AA versions. Starting from any version, multiple new versions can be created. The *derived from* relationship among functional extensions defines a *version tree* that is rooted at some *base version* AA; see Figure 1 for an example. Here we have labeled the derived versions tree indices for clarity.

New AA versions might be defined and installed by network managers, or, ultimately, by any user of the signaling function. However, the latter will require solutions to hard protection and security

Figure 2: AA Version tree example



problems. Some robust authorization mechanism will be required to control which users can use which versions.

Suppose it is the case that every AA version for a given service lies in such a tree, i.e., it can be reasonably considered to be derived from some BV or from another derived version. Then the *derived from* relationship leads to an increment definition of the composition of a AA version: incrementally. Thus, a AA version might be defined by the changes in the set of objects used to define its direct ancestor. Although this incremental approach might lead to a more compact representation of the *AAbase* information, we have not followed this approach in the current ASP EE.

When a new AA version is created, the new class versions it uses will often be extensions (by class inheritance) from earlier versions of the same classes. As a result, the class hierarchy and the AA version hierarchy shown in Figure 1 will often be roughly parallel. This is illustrated in Appendix B, which shows an actual example of the class hierarchy paralleling three AA versions for the specific signaling protocol RSVP.

A.3 Dynamic Class Name Binding

This section shows the coding by which an AA can dynamically map apparent names to definite names and instruct the EE to load the definite-named class over the network.

Suppose that a class needs to construct a new instances of class with apparent name `CC`. Without dynamic binding, one one would write simply

```
CC cc = new CC(x, y, z);
```

With dynamic binding, assume that the AAbase class contains a mapping table in standard format

(see earlier). Then the library call:

```
String CCdef_name = AAcontext.appToDef("CC");

Class c1 = Class.byName(CCdef_name);
Class parm_type1[] = {A.class, B.class, C.class};
Constructor constCC = c1.getConstructor(parm_type1);
```

Assuming that the above code was loaded by the ASP EE's class loader, when the `byName` method is executed, the JVM will invoke the EE's class loader to find the class with definite name. The ASP class loader will then fetch the byte code over the network (if it has not been loaded already) and return a reference of the `Class` class back to the JVM. The JVM will then return the `Class` reference back to the application when returning from the `byName` method.

AA classes can then use the following code to actually construct an instance of the definite class:

```
A a;  B b;  C c;   // Actual parameters

Object parm_values[] = {a, b, c};
CC cc = (CC)constCC.newInstance(parm_values);
```

The reference `constCC` to a constructor object might be computed once when the AA is initialized and then saved in the AA-local data store.

We note that this example could be much simpler if we do not support constructor parameters, but these are an important and useful facility of Java.

Technique A has several disadvantages: (1) it is awkward to explicitly create the formal parameter lists (signatures) and the actual parameter lists, as shown above; (2) it adds bulk and obscurity to AA code; and (3) it may suffer in efficiency.

An alternative coding convention for dynamic binding and loading is less awkward for handling parameter lists and is more opaque and more efficient. This approach uses indirect constructor methods, which we call *constructor proxies*, for each dynamically bound class construction. All of these constructor proxies are by convention gathered into the *AAbase* object.

Using the same example, suppose that the definite name is `"CC1"`. Then we include in the AA a pseudo-constructor method:

```
class AAbase_AA.1 extends AAcontext {
        ...

        // Define constructor proxy methods
        //
        CC newCC(A a, B b, C c) {
                return new CC1(a,b,c);
```

```
        }
        ...
    }
```

Then AA classes would use the following code to construct a `CC` using dynamic binding:

```
CC cc = cntxt.newCC(a,b,c);
```

In this example, the `cntxt` reference is a reference to the `AAbase` class for the current AA.

B Configuring the ASP EE

B.1 EE Configuration

The file `files/asp/asp.conf` configures the ASP EE. An example configuration file is as follows:

```
Debug Remote   bro.isi.edu/9898

##
# Define protocol RSVP
##
Proto   BaseAA "RSVP_base_version"
        DefaultAA "RSVP_confirm_version"
        Channel ChannelAPI E2E 56789
        Channel ChannelRsvpLegacy E2E 5000
```

The syntax of the this file given by the following BNF-like form:

⟨config⟩ ::= { ⟨entry⟩ }
⟨entry⟩ ::= ⟨debug⟩ | ⟨AAEntry⟩ | ⟨commentLine⟩
⟨debug⟩ ::= `Debug` ⟨space⟩ ⟨output⟩ ⟨newline⟩
⟨output⟩ ::= `File` ⟨filename⟩ | Remote ⟨host⟩ `/` ⟨port⟩ | Stdio
⟨AAEntry⟩ ::= `Proto` ⟨space⟩ { ⟨subEntry⟩ ⟨newline⟩ }
⟨subEntry⟩ ::= `BaseAA` ⟨space⟩ ⟨AAname⟩ | `DefaultAA` ⟨space⟩ ⟨defaultAAname⟩
| `Channel` ⟨space⟩ ⟨channelType⟩ | `Bootstrap`
⟨AAname⟩ ::= `"` ⟨allowableChar⟩ { ⟨allowableChar⟩ } `"`
⟨defaultAAname⟩ ::= `"` ⟨allowableChar⟩ { ⟨allowableChar⟩ } `"`
⟨channelType⟩ ::= ⟨channelClass⟩ ⟨space⟩ ⟨TransType⟩ ⟨space⟩ ⟨UDPport⟩
⟨transType⟩ ::= `HBH` | `E2E`
⟨channelClass⟩ ::= any valid fully qualified Java class name that implements `java.lang.Thread`
⟨UDPport⟩ ::= any positive (32-bit) UPD port number
⟨commentLine⟩ ::= `#` { any character except newline }
⟨allowableChar⟩ ::= ⟨letter⟩ | ⟨digit⟩ | `!` | `#` | `$` | `%` | `&` | `=` | `+` | `-` | `_` | `@`

The `Debug` entry specifies where the EE should write any debugging output. By default, the debugging output is printed to the terminal.

The `Proto` entries configure channels for incoming active packets.

- ⟨BaseAA⟩ is the name of the AA that will be used for the initial packet scan to extract an embedded *AAspec*. If this scan does not locate an *AAspec*, then ⟨defaultAAname⟩ is used as the name of the AA, and the rest of the *AAspec* (the *AAbase* name and the path) are obtained from the `AAspecs.conf` file (below).
- ⟨channelClass⟩ is the name of a java class that implements `asp.Channel`.

  There may be multiple `Channel` entries for a single `Proto` entry. The channel entry specifies the name of the java class that will be used (internally by the EE) to open a port and whether packets sent on this channel should be marked as either (1) end-to-end packets or (2) hop-by-hop packets.

  Several classes for ⟨channelClass⟩ are available and may be expanded in the future: `ChannelDatagramV`, `ChannelRsvpLegacy`, `ChannelAPI`. These classes are considered internal to the EE and provide the magic to convert an I/O blocking receive call into an I/O AA upcall.
- `Bootstrap` implies that the AA is to be loaded when the EE boots, before receiving a packet associated with the AA. The EE loads the AA named `DefaultAA`, obtaining the rest of the *AAspec* (the *AAbase* name and the path) from the `AAspecs.conf` file (below).

B.2 AAspecs.conf file

The `files/asp/AAspec.conf` file contains a database of complete *AAspecs*. This database is consulted when an AA is to be loaded and only its name is known, for example, when AAs are loaded from the `asp.conf` file at boot time.

The `AAspec.conf` file contains an arbitrary number of AAspec entries, with each AASpec occupying a single line. The format of the each AAspec is described in the next section.

B.3 AAspec format

An AAspec consists of a variable length text string with the following BNF-like syntax.

⟨AAspec⟩ ::= ⟨AAname⟩ `[` ⟨space⟩ ⟨searchPath⟩ ⟨space⟩ ⟨AAbaseName⟩ `]`
{ ⟨space⟩ ⟨searchPath⟩ ⟨space⟩ ⟨listOfClassNames⟩ }
⟨AAname⟩ ::= " ⟨allowableChar⟩ { ⟨allowableChar⟩ } "
⟨AAbaseName⟩ ::= ⟨className⟩
⟨searchPath⟩ ::= " ⟨innerSearchPath⟩ "
⟨listOfClassNames⟩ ::= " ⟨innerListOfClassNames⟩ "

⟨innerSearchPath⟩ ::= ⟨location⟩ , ⟨innerSearchPath⟩ | ⟨location⟩
⟨URL⟩ ::= BNF syntax as given in [2]
⟨location⟩ ::= ⟨URL⟩ | ⟨privateURL⟩
⟨privateURL⟩ ::= ⟨prefixIP⟩ ⟨IPAddr⟩ | ⟨prefixVNet⟩ ⟨VNetAddr⟩ | ⟨prevHop⟩
⟨prefixIP⟩ ::= `asp-private-ip://`
⟨prefixVNet⟩ ::= `asp-private-vnet://`
⟨prevHop⟩ ::= `asp-private:-ip//PHOP`
⟨IPAddr⟩ ::= any valid hostname or dotted decimal IP address (IPv4 or IPv6)
⟨VNetAddr⟩ ::= any (32-bit) non-negative integer
⟨innerListOfClassNames⟩ ::= ⟨className⟩ | ⟨className⟩ ⟨space⟩ ⟨innerListOfClassNames⟩
⟨className⟩ ::= any valid fully qualified Java class name
⟨allowableChar⟩ ::= ⟨letter⟩ | ⟨digit⟩ | `!` | `#` | `$` | `%` | `&` | `=` | `+` | `-` | `_` | `@`

In the above format, ⟨AAname⟩ is the name of the AA and is assumed to be globally unique (in the world). Once an ⟨AAspec⟩ has been parsed by the EE, the EE keeps an internal representation in a table indexed by ⟨AAname⟩. In future, there may be an authorized unique names authority and corresponding signatures for the code.

The ⟨searchPath⟩ contains a list of code-servers where the EE can fetch the classes for the particular AA. If any class cannot be found at the code-servers in this list then the AA is terminated.

The syntax states that the ⟨searchPath⟩ ⟨AAbaseName⟩ pair are optional. This pair is optional for AAspecs contained in packets but they are not optional for AAspecs in the `AAspecs.conf` file.

The ⟨AAbaseName⟩ is the name of the primordial *AAbase* class, which must extend `AAcontext`. The EE loads and instantiates the *AAbase* class when the corresponding ⟨AAname⟩ is first requested. The EE saves a reference to this instance and maintains a one-to-one correspondence between ⟨AAspec⟩ and ⟨AAbaseName⟩.

The optional pair (⟨searchPath⟩, ⟨listOfClassNames⟩) enables certain classes to be found using a different search path. This feature allows developers to extend an AA and have the extension classes reside in a code-server different from the base AA classes.

An example of an AAspec for an RSVP AA might look like the following string. Note that the newline is for presentation purposes only, as there are no newline characters in AAspecs.

```
"RSVP_base_version" "asp-private-ip://PHOP,asp-private-ip://128.9.160.128,asp-private-vnet://1"
        "rsvp.VersionBase"
```

B.4 VNet Configuration

This section describes the required configuration files when the EE provides VNet support.

The VNet network stack uses two configuration files, one to store initial network interfaces and the

other to store initial route entries. The ASP EE reads these files during the EE boot sequence. The configuration information is converted into an internal structure, which may change with routing updates, for example. Currently, the EE does not modify these configuration files if the internal structure changes. This means that when the EE is restarted, it cannot restore the internal routing information when the EE terminated.

The names of the VNet configuration files must be located in the `files/asp/InterfaceTable.txt` and `files/asp/RouteTable.txt` when the EE is started.

We shall first provide an example of the contents of these files before specifying the syntax. The following interface table is for the machine `bro.isi.edu`.

```
asp.InterfaceVN lib.AddressUI 1 local/1111 2 obelix.cs.sun.ac.za/2222
asp.InterfaceVN lib.AddressUI 6 local/1112 3 son.isi.edu/2222
asp.InterfaceVN lib.AddressUI 7 local/1113 4 son/2224
asp.InterfaceVN lib.AddressUI 8 local/1120 5 207.3.230.162/2222
asp.InterfaceVN lib.AddressAnep 11 bro.isi.edu/22222/15 12 son.isi.edu/22222/15
```

Each entry specifies a virtual point-to-point link in the virtual topology. Each link is specified by two end-points, which are in turn specified by an (IP address, UDP port) pair. The end-point of the first link is `local/1111` and `obelix.cs.sun.ac.za/2222`. The keyword `local` means the local machine and the numbers are UDP port numbers. The integers preceding the end-point address are the VNet addresses, which are unique for each (IP address, UDP port) pair. Note that there may be multiple VNet address associated with a single IP address, as is the case with machine `son`.

The following example route table contains 5 route entries:

```
asp.RouteVN 1 1
asp.RouteVN 1 2
asp.RouteVN 6 3
asp.RouteVN 7 4
```

The integers are VNet addresses as defined in the interface table above. The first entry specifies the loopback route which must be explicitly listed if loopback functionality is required. The remaining entries specify individual routes to each remote VNet address from the local VNet address.

The syntax of these files are defined in a BNF-like form, as follows.

⟨interfaceTable⟩ ::= { ⟨infEntry⟩ ⟨newline⟩ | ⟨commentLine⟩ ⟨newline⟩ }
⟨infEntry⟩ ::= ⟨infClass⟩ ⟨space⟩ ⟨addrClass⟩ ⟨space⟩ ⟨localVNetAddr⟩
⟨space⟩ ⟨localUDPAddr⟩ ⟨space⟩ ⟨remoteVNetAddr⟩
⟨space⟩ ⟨remoteUDPAddr⟩
⟨infClass ⟩ ::= any java class implementing `asp.InterfaceL3`
⟨addrClass⟩ ::= any java class implementing `lib.AddressVNet`
⟨localVNetAddr⟩ ::= ⟨VNetAddr⟩
⟨remoteVNetAddr⟩ ::= ⟨VNetAddr⟩

bit	message type
0	network I/O
1	class loading
2	interaction with AAs
3	routing notifications
4	unused
5	timers
6	kernel traffic control
7	process scheduling

Table 1: EE debugging bit mask

logging-level	meaning
0	always
2	critical
3	error
4	warning
5	notice
6	informational
7	debug
8	include hex dumps

Table 2: EE debugging bit mask

⟨VNetAddr⟩ ::= any (32-bit) positive integer
⟨localUDPAddr⟩ ::= ⟨IPAddr⟩ / ⟨UDPport⟩ | `local/` ⟨UDPport⟩
⟨remoteUDPAddr⟩ ::= ⟨IPAddr⟩ / ⟨UDPport⟩
⟨IPAddr⟩ ::= any valid hostname or dotted decimal IP address (IPv4 or IPv6)
⟨UDPport⟩ ::= any positive (32-bit) UPD port number
⟨commentLine⟩ ::= `#` { any character except newline }

⟨routeTable⟩ ::= { ⟨routeEntry⟩ ⟨newline⟩ | ⟨commentLine ⟨newline }
⟨routeEntry⟩ ::= ⟨routeClass⟩ ⟨space⟩ ⟨localVNetAddr⟩ ⟨space⟩ ⟨remoteVNetAddr⟩
⟨routeClass⟩ ::= any Java class implementing `asp.RouteL3`
⟨localVNetAddr⟩ ::= ⟨VNetAddr⟩
⟨remoteVNetAddr⟩ ::= ⟨VNetAddr⟩

C Running the ASP EE

C.1 Command-line invocation

The following command may be used to start the EE from the command line.

```
Java asp.AspMain [-d debug-mask] [-l logging-level]
        [-f alternative-asp.conf]
```

The `-d` option sets the internal debug mask, which is a 32-bit mask that defines which types of debugging messages are logged and which are not. By default all messages are logged. Table 1 shows the bit pattern of the mask and how each bit corresponds to the type of message. For example, if one wants to log network I/O and routing updates, one would use the option `-d 9`, which is obtained from $2^0 + 2^3$.

The `-l` option sets the EE's logging-level, which specifies a threshold above which messages should not be logged. For example, if the logging-level is L, then the EE does not log any messages that have logging-levels higher than L. The available logging-levels are defined by the `lib.Debug` class and are shown in table 2.

Finally, `-f` option specifies a replacement for the configuration file `asp.conf`. See section B.1 for details.

C.2 AnetD invocation

Starting the EE over Anetd requires two steps. First, all the support files must be transferred to each node. (The support files include `files/asp/asp.conf`, `files/asp/AAspecs.conf` and the VNet interface and route tables.) The following Anetd command shows how the `asp.conf` given as an URL on machine `bro`, is transferred to a file `files/asp/asp.conf` on machine `son` (the `C=` option is merely a comment regarding this execution):

```
aload son.isi.edu F=http://bro.isi.edu/ARP/files/asp/asp.conf D=asp \
        C=Copying_asp.conf
```

Second, the EE must be invoked remotely, again using Anetd commands. The following Anetd command shows how the EE, given as an URL on machine `bro`, is started on `son` (notice the "~" symbol in the `J=` URL cannot be ommitted, because it represents the fully qualified name of the Java class):

```
aload son.isi.edu J=http://bro.isi.edu/ARP/~asp/AspMain D= T=15 \
        O=OUTPUT R=ERRORO C=ASP_EE
```

C.3 System Properties

During EE boot time, the EE checks whether the Java system property `NATIVE` is defined. If the propery is defined then the EE calls native methods to provide a richer set of I/O primitives than the JDK. If this property is not set then the EE does not use any native methods.

System properties are typically set from the command line when invoking the JVM. For example, to define the `NATIVE` system property when starting the EE, use the following command:

```
java asp.AspMain -DNATIVE
```

References

[1] D. Scott Alexander, Bob Braden, Carl A. Gunter, Alden W. Jackson, Angelos D. Keromytis, Gary J. Minden, and David Wetherall. Active network encapsulation protocol (ANEP). `ftp://www.cis.upenn.edu/pub/switchware/public_html/ANEP/index.html`, 1999.

[2] Internet Engineering Task Force. BNF for specific URL schemes. `http://www.w3.org/Addressing/URL/5_BNF.html`.