Introduction to the ASP Execution Environment (v1.6)

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

USC/Information Science Institute
{isi-arp}@isi.edu

February 18, 2003

Contents

1 Introduction

The ARP (Active Reservation Protocol) project [1] at ISI is developing an active networks framework for the implementation and deployment of complex network control-plane functions, such as network signaling and management. However, the techniques being developed should be generally applicable to any programmable network element.

This framework is embodied in the *ASP Execution Environment* (ASP EE), which supports the dynamic installation and execution of *active applications* (AAs). [2] The ASP EE and its AAs are expressed in Java byte code, which provides a (nearly) platform-independent encoding that supports dynamic loading of AAs.

1.1 Active Node Architecture

The ASP EE follows the node reference model [3] that has been developed by the active networking research community. This hierarchical model for organizing the software within an active node is illustrated in Figure 1.

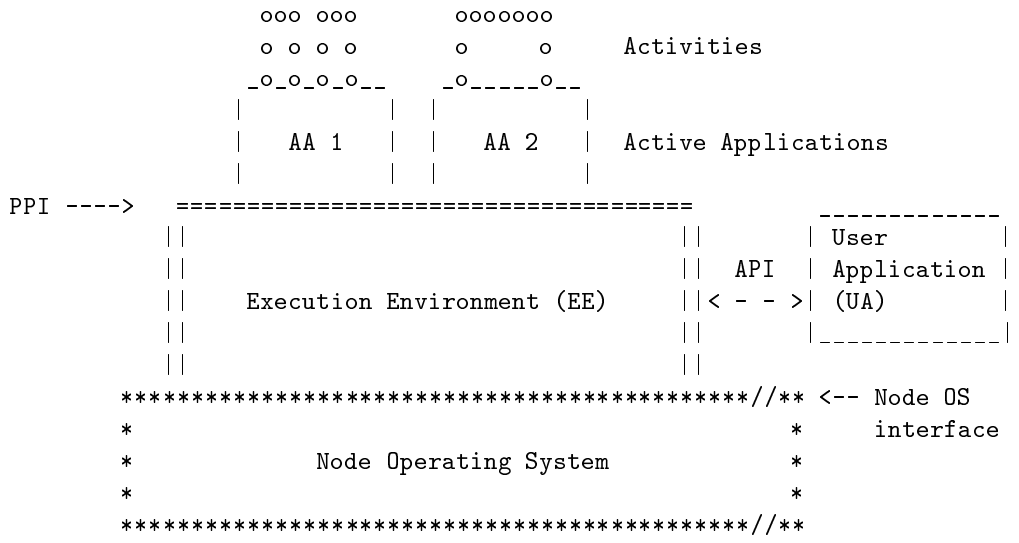


Figure 1: Active Node Architecture

An active node is under control of an operating system known as the *node OS*, which is expected to allow simultaneous execution of multiple *Execution Environments* (EEs). The node OS mediates

[1] DARPA contract number DABT63-97-C-0049.

[2] "ASP" stands for *active signaling protocol*, even though the ASP EE is designed for a wide variety of network control applications in addition to signaling.

among EEs and isolates 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 protect the node by enforcing boundary and resource limits on each of its EEs.

Each *active application* (AA) is written for execution within a particular EE. An AA may be highly dynamic, while an EE is expected to be a stable feature of the node. Each EE should be capable of supporting multiple simultaneous AAs. The EE should be thought of as extending the node OS "upwards" into user space.

The ASP EE is effectively a user-level operating system to control AA execution. The ASP EE supports a defined interface to EE services that we call the *protocol programming interface* or PPI. The PPI is described in a companion document [6].

The ASP EE is designed to support both persistent AAs and transient AAs. For example, an active "ping" program might install a transient AA that executed once and was finished, while signaling and routing applications will generally be persistent. A persistent AA may support multiple independent signaling or control *activities*, as suggested 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 typically supports only a single activity, the routing computation to build the shared forwarding database in the node. Each multi-activity AA may have its own rules for resource sharing, multiplexing, and isolation among the different activities. A persistent AA will typically maintain persistent state that is partitioned among its current activities.

The ASP EE assumes that active activities are launched by *user applications* (UAs). A UA might be an end-user application executing within the normal operating environment of the user's end system, or it might be a proxy or other control program in a network management station, for example.

1.2 EE Functions

As an operating environment for active applications, an EE must 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. However, the code to implement real network control algorithms is typically too large for an individual capsule, so the ASP EE supports only out-of-band loading.

- Security and Resource Protection.

  An 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, and in fact the

ultimate goal of the ASP EE is safe execution of arbitrary user-supplied (and hence completely untrusted) AAs. Once an AA has been injected into the network, it should always be possible to identify its source in case it misbehaves.

- Network I/O.

  The node OS and the EE must cooperate to dispatch received active packets to the appropriate AAs and to allow these AAs to send packets into the network.

- UA/AA API

  As illustrated in Figure 1, an EE in an end system must support an API that a UA can use to launch an active activity in a specific AA. This API will probably use some form of local interprocess communication.

- Timing Service

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

- 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.

2 Features of the ASP EE

The ASP EE currently supports all of the general EE features presented in the preceding section; that is, it supports dynamic loading of AA code, security and resource protection, network I/O, a UA/AA API, a timing service, and inter-AA communication. Because the ASP EE is designed for activating complex network control protocols, it supports the following additional features:

- Sharable AA code,
- Dynamic class binding, and
- Soft-state containers.

These features are discussed in the remainder of this section. In the rest of this document, an unqualified "AA" or "EE" refers specifically to an ASP AA or an ASP EE, respectively.

2.1 Dynamic Loading

It is possible to configure the ASP EE to load an AA at boot time; however, the real power of active networking arises from dynamically loading AAs from remote nodes. We use the term *active packet*

for a packet whose arrival will cause the ASP EE to dynamically load the appropriate AA, if it is not already loaded.

An active packet must contain or imply a data object to control dynamic loading. In the ASP EE, this data object is called the *AAspec* and consists of a variable-length text string in a format is defined in Appendix A. An AAspec contains three pieces of information:

- a globally unique name for the AA;
- the name of a primordial Java class, specific to the AA and known as the *AAbase* (Section 2.6); and
- a search path specifying one or more code servers from which classes that compose the AA can be fetched.

  The search path can include explicit URLs for code servers and/or an implicit reference to the previous-hop node from which the packet arrived.

The node OS demuxes incoming packets to the ASP EE. This demuxing step might be controlled by the *type ID* field in the ANEP header [1] or by some other packet filter criterion applied to the input stream. [3] [3]. The ASP EE in turn determines an AAspec for the packet and uses it to demux the packet to the appropriate AA, loading that AA if it is not yet loaded, using the search path from the AAspec. The EE finally passes the active packet to the loaded AA as input.

There are four ways that the ASP EE can determine the AAspec from an incoming active packet. The AAspec may be:

1. carried explicitly in an ASP header prefixing the AA payload, or
2. defined implicitly by an EE-specific packet filter that maps a subset of incoming packets into a pre-configured AAspec, or
3. carried explicitly within the AA payload using some AA-specific encapsulation syntax, or
4. defined implicitly as the AAspec of the AA whose packet filter captured the packet.

Case 4 can occur only after the AA has already been found and loaded, which requires one of cases 1 - 3. The choice among cases 1 - 3 is controlled by packet filters file named `asp.conf` (see Appendix of [6]). Cases 1 - 3 are respectively determined by configuration lines of the form:

```
InChannel <filter spec> invokes *                          [Case 1]

InChannel <filter spec> invokes AAname <AAname>            [Case 2]
```

[3] presumably under some security limitations

```
InChannel <filter spec> invokes Legacy <AAname>          [Case 3]
```

In each case, `<filter spec>` defines a packet filter to select some subset of the traffic demuxed to the ASP EE. Filter specs are defined in Section 2.2. In cases 2 and 3, these configuration rules yield an AAname, and another ASP configuration file is used to map the AAname into a complete AAspec.

Case 3, burying the AAspec within the payload using an AA-specific encapsulation, is useful for interworking with legacy non-active protocols. For example, to support interworking with standard RSVP the AAspec can be buried in an opaque RSVP object, which will be ignored but forwarded by legacy RSVP in non-active nodes. 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, which is needed to load the AA. This circularity is avoided by mapping the packet to the AAspec for a service-specific preamble AA, which is used only to extract the desired AAspec from the AA payload.

Note that the last two items in the AAspec – the AAbase class name and the search path – are needed only for loading the AA code. Once the code has been loaded, subsequent protocol packets for the same AA can be demultiplexed using only the AAname field of the AAspec. However, an AA may not be able to predict the order in which packets will arrive from different sources. Furthermore, the AA code, once loaded, is only cached; it might time out and be replaced in memory during a pause in an AAs activities. For robustness and simplicity, therefore, the ASP EE requires that the AAspec be determinable, by one of the cases above, for every protocol packet that is received. An exception is AAs that are configured to be loaded at boot time.

2.2 Network I/O

2.2.1 Network Access Modes

The ASP EE supports two network access modes for active networking.

- *Virtual connectivity*

  In virtual connectivity mode, active nodes are interconnected by *virtual links* formed by UDP/IP tunnels. The IP address and the UDP destination port of a tunnel endpoint effectively define a *virtual link layer (VLL) address* for the EE in that node. Each virtual link to another node then defines a *virtual interface*.

  Since realistic networking research generally considers internetworking. Therefore, in the virtual connectivity mode active node is assigned a *virtual link layer address*, and the EE or AA should use these addresses to provide a default forwarding mechanism between VLL interfaces. These virtual network-layer addresses may occupy a new address family, although as a special

case they might in fact have the format of IP addresses. The active node must be able to map virtual network addresses into VLL addresses to send active packets to other nodes.

An ASP AA operating in virtual connectivity mode may build its own 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. VNET implements instances of layers 2, 3, and 4 of the OSI protocol reference model, i.e, the link, (inter-)network, and transport layers. In the internetwork layer, VNET in the ASP EE provides unicast default forwarding in the virtual internetwork. VNET is described more fully in Section 4.3.

Although VNET currently implements a virtual link layer, it would be possible in principle for VNET to also provide direct access to some real hardware link layer. This has not been implemented.

- *Native IP connectivity*

  This mode assumes that active networking is an extension of the Internet protocol suite, providing AAs with direct access to the Internet protocol stack. Native IP connectivity is supported by an ASP EE component called *INET*. It includes access to packet payloads after processing by legacy protocols such as TCP and UDP. There is also support in ASP for diversion of active packets and non-active packets from the native IP stream. [4]

2.2.2 The Channel Interface Abstraction

The ASP EE's network I/O model is generally consistent with the *channel* abstraction of the EE/NodeOS reference interface [4]. The channel abstraction provides a flexible interface for demultiplexing incoming packets using packet filters and for sending packets with appropriate encapsulation. It optionally allows use of the default protocol stack that is internal to the node OS.

A running AA may have packets diverted to it as an interceptor or as an endpoint. In either case, packets are selected by the node OS based on a set of selection criteria communicated to it by the channel abstraction. ASP allows an AA to intercept and retransmit these packets transparently, that is, with the source and destination address unchanged. The AA may manipulate the packet contents, e.g. it may transcode video in the payload, but the packet still seems to have originated from the sender. Alternatively, the AA may be an endpoint, consuming a packet addressed to it or starting new connections as an endpoint. ASP supports both types of AAs.

The ASP EE currently implements input channels (InChannels) and output channels (OutChannels). An InChannel specifies a packet filter to select a subset of the input stream and present it to the EE, and perhaps the AA, that created the channel. OutChannels allows an AA to send active and inactive packets.

The node OS document [4] specifies syntax and semantics for three parameters that define a channel –

[4] Earlier versions of the ASP EE had native IP support that provided access to raw IP using native methods. This support has been replaced by a unified interface using the channel abstraction.

the *protocolSpec*, the *addressSpec*, and the *demultiplexKey*. For an InChannel, these three parameters define the `<filter spec>` used to capture incoming packets.

- The protocolSpec parameter is a character string that defines a stack of protocols that will process a selected incoming packet. The headers corresponding to these layers will be removed from the packet, which will then be delivered to the AA to which the channel is bound. The PPI uses essentially the same syntax and semantics for its protocolSpec.
- The addressSpec parameter is a character string that defines specific demultiplexing fields in the protocols listed. This parameter is replaced by an specialization of `asp.net.Attrib` object (essentially, a struct) containing binary values.
- The demultiplexKey parameter specifies binary matching values to do arbitrary filtering of input packets selected by the addressSpec. The demultiplex key may contain one or more (header, offset, length, value) tuples.

  The demultiplexKey is supported for PPI InChannels that use VNET and the raw IPv4 channel. Support for the demultiplexKey under native IP operation for UDP channels is a future task.

The ASP EE implements three kinds of channels.

- ASP EE InChannel

  ASP EE InChannels implement the `InChannel` entries found in the EE's `asp.conf` configuration file, as described earlier in Section 2.1. An `InChannel` entry in that configuration file defines a filter that determines what packets will be received by an ASP EE and also how an AAspec for that AA is to be extracted from these packets. At EE boot time, an EE InChannel will be opened for each line in the configuration file, to intercept packets with the specified `<filter spec>`. EE InChannels are never closed.

- Implicitly-opened AA InChannel

  When a packet is intercepted by an ASP EE InChannel, the EE passes the packet to the appropriate AA through an *implicitly-opened* AA InChannel. For a dynamically-loaded AA, the first packet received through this implicit channel will be the packet that triggered loading of the AA. An implicitly-opened AA InChannel will have the attributes of the ASP EE InChannel that intercepted the packet: its `<filter spec>` parameters will match those of the configuration file entry that created the EE InChannel.

  An implicitly-opened AA InChannel cannot be closed. It is possible for an AA to have multiple implicitly-opened InChannels, as the result of multiple `asp.conf` entries specifying the same AA.

- Explicitly-Opened AA Channels

  An AA explicitly opens and closes OutChannels, which it needs to send packets. It can also explicitly open additional InChannels.

AA channels form part of the PPI ([6]. The ASP EE's PPI supports several extensions to the node OS channel abstraction.

- The PPI channels are used for communication across the API with a local User Application (UA), for inter-AA communication, as well as for network I/O.
- The PPI supports stream-based I/O in addition to the datagram-based I/O defined in the current node OS spec. API channels are always stream-based. We also support the reliable transport protocol RDP under VNET.
- PPI channels are opened implicitly by the ASP EE as well as explicitly by AAs, as explained earlier.
- When an AA receives a packet via an upcall, it also receives an `asp.net.Attrib` object that defines the addressing/demultiplexing attributes of the packet, including in particular the actual values that matched wildcards in the channel constructor.
- The protocolSpec used in the PPI extends into the EE protocol stack and can optionally include the processing of the AAspec.

It should be noted that the signaling applications envisioned for the ASP EE may require fine-grain control over network I/O. It is possible that an ASP EE will require a superset of the channel functions finally adopted for the reference EE/NodeOS interface. For example, an ASP AA may need to:

- Learn the network interface (real or virtual) on which the packet was received and explicitly control the interface on which a packet is sent, for a unicast or multicast destination.
- Learn the network-layer source address with which a packet arrived and set the source address with which a packet is sent (not necessarily to one of its own interfaces).
- Learn the IP TTL with which a packet arrived and set the IP TTL with which a packet is sent.
- Send a packet with the Router Alert option or analogous mechanism.
- Learn the set of interfaces on the node.

2.2.3 User Application API

The UA/AA API is implemented in the ASP EE using the channel abstraction. This unifies the I/O functions of an AA; for example, when a UA's request arrives at the UA/AA interface, the EE can determine the AAspec for the target AA using any of the methods listed in Section 2.1, and using the same configuration files.

However, an API channel differs in several ways from a network I/O channel.

- API channels do not correspond to network interfaces.
- API channels are only stream-oriented (since they are implemented using TCP).
- The AA cannot explicitly open an API channel.
- Implicitly-opened API InChannels are created dynamically by the EE. A new API channel is created when a UA opens a TCP connection to `<TCP server port>` and sends the first message.
- API InChannels and OutChannels are identical. To send a reply to a UA, the AA casts the implicitly-opened InChannel into an OutChannel in a call to `sendPacket()`.
- An API InChannel must be able to provide an *end-of-file* signal to the AA to indicate that the UA has closed the connection.

2.3 Inter AA Communication API

The Inter AA communication is implemented in the ASP EE using the channel abstraction. An Inter AA channel differs in several ways from a network I/O channel.

- Inter AA channels are unidirectional pipes, only the AA that opened the channel can write into that channel. Thus, two channels are required, one in each direction, for creating a full duplex connection between two AAs.
- The AASpec of the reader AA must be specified while creating the Inter AA channel.
- The reader AA can reject the Inter AA channel by calling close(), the writer AA will then receive an exception on the next write.

2.4 Security and Isolation

The ASP EE provides the following security features.

- EE Protection

  The ASP EE is be able to protect itself from AAs. In addition, a malicious AA is prevented from circumventing the EE's mediation between applications and resources.

- AA Isolation

  The AA provides a natural unit for data isolation in the ASP EE. The EE therefore isolates each AA, preventing it from committing either boundary violations or resource violations. 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 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 ASP EE is believed to provide a high degree of protection against boundary violations by general (unprivileged) AAs. The basic ASP EE mechanism for preventing boundary violations is based on 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, as described in Section 3.5.

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 – memory and network output bandwidth – are not currently protected. In any case, we should note that it is difficult to completely eliminate the interactions caused by shared resources.

- Prevention of AA spoofing

  An unprivileged AA is prevented by the EE from sending with an AAspec not its own.

- Configurable AA Capabilities

  The ASP EE defines AA privileges in terms of well defined capabilities. A good example of these concepts comes from a routing protocol AA, which needs to perform privileged operations to modify the EE routing tables during its execution. The EE must have a mechanism that allows a routing protocol AA to make these changes but disallows accidental or malicious tampering of the routing tables by other AAs.

The current capabilities defined in ASP are as follows:

1. interface - The privilege to modify the EE network interface table.
2. route - The privilege to modify the EE network route table.
3. divert - The privilege to perform arbitrary packet interception.
4. process - The privilege to terminate another AA.
5. file - The privilege to read and write outside the file space of an AA.
6. socket - The privilege to use the JDK sockets interface rather than using the channel based PPI. Useful for legacy or third party software.
7. native - The privilege to load native code into the JVM
8. noaaspec - The privilege to suppress the mandatory AAspec generated by the EE in outgoing packets.

Capabilities are attached to AAs using an EE configuration file, which is consulted at the time of AA loading. In the future, it should be possible to provide these capabilities over the network using a secure distribution mechanism. The long range view is that the ASP EE will incorporate a security architecture and distribution mechanism currently under development other security oriented groups within the Active Networks research community.

There are still major limitations of the current security mechanism of the ASP EE.

- It does not support either hop-by-hop or end-to-end authentication of packets or user certification.
- Hence, AA capabilities cannot be tied to the user or user group that initiated the active packet.
- AA privileges are controlled by AAname, but there is no mechanism to prevent an intruder sending an AAspec with an arbitrary path but a defined AAname.

2.5 Sharable AA Code

The ASP EE supports the sharing of common byte code among different AAs, for reasons that are detailed in the next section. However, this this feature is optional because of its implications for unloading AAs and because of other technical problems it raises for AAs.

To understand the problem of unloading AAs, we must understand the role of the ClassLoader in Java. Class loaders are responsible for loading code into the JVM. By definition, each class loader creates a separate name space for the code and data of the objects it loads. Two different class loaders can load distinct copies of the same class with the same fully qualified class name. Essentially, each fully qualified name of a loaded class is implicitly prefixed by a reference to the ClassLoader object that loaded the class.

The only way to unload byte code, removing it from the heap, is to remove the ClassLoader that was used to load the class. This would naturally lead to the idea of using a single ClassLoader per AA while sharing common byte code across different ClassLoaders. This is possible if this class loader explicitly shares class references among different loader instances by means of a global table. Unfortunately, this approach would interact in unpleasant ways with Java's access control rules, for reasons detailed in Section 3.3. The preferred way to share byte code among a set of AAs is to use a single ClassLoader for all the AAs.

These considerations create a conflict between the ability to unload an individual AA and the ability to share byte code.

At the present time, the ASP EE can be operated in one of two modes, code-sharing or non-code-sharing. In the code sharing mode, it uses a single ClassLoader and shares common byte code. This mode is expected to be useful only for production operations where there is a lot of overlap among large, complex AAs. In the other mode, there will be a distinct ClassLoader for each AA, with no sharing. This mode will be useful when there is significant churn of AA code being installed and removed, e.g., for testing.

The present two-state scheme could be generalized to define sharing groups, with a ClassLoader per group. A particular AA could be assigned to a group, and perhaps to a group of one ClassLoader. The AAspec would specify the loader group. Further development in this area is desirable.

2.5.1 The Case for Sharing AA Code

The code sharing mechanism of the ASP EE was designed to support a large number of different AAs, including AAs for multiple versions of the same service, for the following reasons.

- The real-world application of active networking will lead to new control and management services.
- For the more complex services, there will be a proliferation of different versions of each service, with each version implemented as a distinct AA. Protocol updates may introduce new features or correct errors in earlier versions of the protocol.
- Some complex control-plane protocols have optional features, which leads to much complexity in protocol and implementation. 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. Such protocols are evolving entities to which new features are often added.

AAs for real signaling protocols (e.g., RSVP) are large and complex, but different AAs for the same service will often share much common code. To significantly reduce the memory footprint for such active applications, the ASP EE supports the sharing of common class byte code among different AAs.

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 common methods and fields are available in Cy, but their byte code need be loaded only once.

2.5.2 The Consequences of Code Sharing

The sharable code capability introduced several technical problems for the ASP EE

- Java class definitions may contain embedded data with the `static` attribute, or they may use *static initialization* to set data values when a class is loaded. In either case, the use of the Java `static` attribute creates data values that can be seen and perhaps changed by any AA sharing the same byte code. This violates the isolation of these AAs, and hence it not permissible. The result is to severely restrict the usefulness of the `static` attribute in AAs that share code under the ASP EE.

  Data that is global to the classes within the AA but local to the AA is needed to provide *anchors* for locating variable object references local to the AA. Static data is normally used in Java to provide a generalized form of global data that can be accessed from multiple instances of a class, so static data would normally be used for such anchors. When static data is not suitable within the ASP EE, the EE provides a new mechanism that supports *AA-local data* (*AA-LD*) to replace static variables and static initializers. The AA-LD mechanism is described in Section 3.2.

- To make shared code useful for network control protocols, the ASP EE supports dynamic binding of class names. Suppose that a method M of a shared class C constructs a new instance of some other class C'. The particular name of the C' class to be constructed depends upon which particular AA is executing M. This situation arises when two AAs share common class code C but use different versions of class C'.

  To handle this case, the class name of a class to be constructed needs to be *dynamically bound*. Dynamic class name binding also adds a versioning mechanism to ASP's dynamic loading. The following Section 2.6 describes how such dynamic class name binding is achieved within 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.

2.6 Dynamic Class Name Binding

In dynamic class name binding, an *apparent class 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 the Java constructor operator `new` with an apparent class name.

Dynamic class name binding in an ASP AA can be logically divided into two distinct steps that we call *name mapping* and *version resolution*.

1. Name Mapping

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

2. Version Resolution

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

   In many operating systems, dynamic loaders can load code fragments into running applications. These code fragments reside in dynamically loadable libraries, and these libraries adopt a standardized 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 of the ASP EE adds version resolution support. In particular, it supports dynamic resolution of a *wildcard* version specification to find the "latest" version of the code[5].

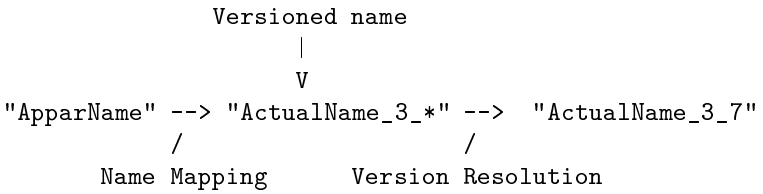
[5] In object-oriented terminology, there are actually two forms of versioning, class versioning and instance versioning.

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 class name and the same major version number are assumed to be compatible in both interface and function. The minor version number is stepped for a new implementation that does not change the interface or functional compatibility 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[6].

In summary, dynamic class name binding in the ASP EE first maps an apparent name into a *versioned name*, and then resolves the versioned name into an actual class name to be loaded. This is an example of this two-step mapping:



The following process is used for dynamic class name binding.

- The primordial *AAbase* class contains the AA's map for dynamic mapping of apparent names to a versioned name. The procedure for this mapping is in principle AA-specific, but an example is given below.

  In the *AAbase* class, one could create a `java.util.Hashtable` object to map the apparent name to versioned name,

  ```
  Hashtable nameMapper = new Hashtable();
  nameMapper.put("ApparentName", "VersionedName");
  ```

  Then in other portions of the AA code, there would be usage of a string containing the apparent name, and it could then map it to the versioned name by using,

  ```
  // apparname is a String received by the AAbase class
  String versionname = (String)nameMapper.get(apparname);
  ```

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.

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

Name mapping is described more fully in a companion document "How to Write an ASP AA" [6].

- The AA then passes the versioned name to the ASP EE routine `AspSystem.forName(versionname)` that performs version resolution (i.e., resolving a wildcard), fetches the resulting actual class, and loads the class code.
- The AA then uses the Java reflection interface to construct an instance of the actual class. This is somewhat complex if there are constructor parameters. However, the companion document "How to write an AA" [6] 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.

It would have been possible to perform both name mapping and version resolution entirely inside an AA, allowing different AA programmers to use different conventions. However, resolution of a wildcard version specification within the AA would require two round trips to the code server, the first to return all available version names to the AA, and the second to load the specific version picked by the application. We therefore decided to perform version resolution in the ASP EE, which will allow one round-trip time to be saved by moving wildcard resolution to the code servers. The disadvantage of this choice is that it requires a single convention for version specification (shown above).

2.7 State Containers

The ASP EE has a state repository mechanism that allows each AA to maintain multiple private name-to-object mappings or "tuple spaces". This mechanism supports soft-state as well as hard-state. Soft state is discarded if it is not periodically refreshed, while hard-state must be explicitly removed. In contrast, the soft-state approach has two advantages, simplicity and robustness. Managing soft-state is simpler because one operation can be used to *create*, *refresh*, and *modify* state. Robustness is improved because hardware and software failures cannot result in dangling state, and because the repeated refresh messages can adapt to external changes such as new network routes. The ASP EE's tuple space mechanism provides timers to support soft state.

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 corresponding 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, using AA-LD for saving context and AA threads for event management. 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. An 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.

3.1 Processes and Threads

The ASP EE implements a simple Java-based process model to control execution of multiple AAs. The model includes the definition of a process, a rudimentary process scheduler, and logically separate data spaces for different AAs. The ASP EE implements each AA with a separate process, so there is a one-to-one correspondence between processes and AA executions in the node. An ASP process is defined by subclassing the AspProcess class. A single process may have zero or more persistent Java threads, which are realized using the AspThread class.

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.

The ASP 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.

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 for a preemptive strategy to guarantee that all threads at the same priority are given a chance to run. In addition to preventing starvation, the ASP EE 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.

An ASP thread is created 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 the currently executing thread. This method is similar to the function of the same name 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 Java 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 AAs. Two threads that are part of the same Java `ThreadGroup` and handed off to two separate AAs 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, allowing this thread to be handed off to an AA in isolation from the threads used by other AAs.

3.2 AA Local Data

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 (or AA-LD) for variables that are global within the AA but local to it.

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 data space 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.

3.3 Class Loading

The ASP EE fetches AA classes from code servers using a search path found in an AAspec for the AA. During the execution of the AA, references to additional classes cause the JVM to invoke a loader method of the appropriate ClassLoader. This is the ASP ClassLoader, which overrides the standard JVM ClassLoader.

The upcall from the JVM provides the name of the needed class, and an stack examination by the ClassLoader provides implicit information about which application was executing when the load request occurred. This information is used to map the load request back to the originating AA and its AAspec, providing the necessary search path to find the byte code and load it.

This section discusses the design considerations behind the ASP EE support for loading AA code, sharing common code among different AAs, and applying the appropriate search path to each individual class loading request.

Although class loaders create individual name spaces, it is possible to share classes across multiple class loaders if the class loader explicitly shares class references between different loader instances by means of a global table. In this case, the fully qualified name of a class remains prefixed with the class loader instance that performed the original load operation.

When the ASP EE is executed in the non-code-sharing mode, each AA uses a separate instance of a class loader and maintains its own separate copy of all class code. An instance of a class loader for a particular AA uses 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 can maintain a search path as member data and use this search path for all loading requests.

To support the sharing of class code among applications, 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 maintain a class loader instance per AA, analogous to the Java Applet model, but 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 populate a shared table that is accessible to the different instances of class loaders.

  Under this approach, 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, symbols from two different classes are accessible to each other if they are in the same package and the symbols are qualified with 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 be a one-to-one correspondence between AAs 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 by 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 is 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.

In its sharable-code mode, the ASP EE implements the last (single class loader) method. This provides the necessary functionality but avoids the complexity of implementing cooperation among multiple class loader instances.

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`.
2. No static class data is allowed; AAs must use AA-local data instead. 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.

3.4 The AAbase Class

For every AA, there is a primordial class known as the *AAbase* class. This class must extend the EE's *asp.AAContext* class. Its Java class name appears in the AAspec for the AA. When the ASP EE begins loading a new AA, it invokes the ASP class loader (Section 3.3) to fetch and load the *AAbase* class, and instantiates this class. 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 4.1.
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 2.6) or proxy constructors (see the next section and [6]).

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`).

3.5 Security and Isolation

3.6 Protection of the EE

The Java package mechanism in conjunction with Java access rules provides coarse-grained protection for the EE. Portions of the EE that should be inaccessible from an AA are protected by these static language mechanisms when applicable. For those portions of the EE that need dynamic protection mechanisms, the ASP EE installs a custom Java Security Manager to block illegal access requests from an AA for either JDK- or EE-restricted functions.

3.7 AA Isolation

Each unique *AAspec* defines a separate AA instance that is isolated from other AA instances. One AA cannot access the state of a second AA that either is running concurrently or has left behind some state from a previous epoch.

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 (Section 3.3).

Because all AAs run in a single address space, isolation of an AA requires that references to its objects cannot leak to other AAs. In addition, references that an AA passes to the EE across the PPI must not be leaked to other AAs. This is accomplished by passing all parameters across the PPI by value, i.e., as a reference to a cloned copy of the parameter object. This cloning is done by the callee (e.g. the EE) and is transparent to the caller. The classes of all parameters are scoped `final`, to prevent a user overloading the clone method and bypassing the copy operation.

4 Network I/O

4.1 Network I/O Primitives

The ASP EE's Protocol Programming Interface (PPI) implements network and API I/O using two basic calls: *sendPacket()* and *receivePacket()*.

Each AA InChannel is bound to the particular AA to which it will deliver the payloads of the packets selected by the channel parameters. The EE opens an implicit InChannel when it loads a new AA; an AA may optionally open additional InChannels once it gains control of the CPU.

Conceptually, each InChannel applies a packet filter to the packet stream and intercepts matching packets. It may do well-known protocol processing on these packets. It then passes the remaining payload up to the AA to which it is bound. The node OS protocolSpec determines what protocol processing takes place, while the addressSpec and demultiplexKey determine the filter.

Like the node OS interface, the ASP EE uses an *upcall* model for passing incoming packets to AAs. The EE invokes the following method in the *AAbase* object:

```
void     receivePacket(InChannel chan, NetBuffer msg,
                       Attrib attrib) throws IOException;
```

The `chan` parameter specifies the network or API input channel through which the packet arrived, while the `msg` parameter specifies a buffer. The `attrib` parameter conveys an extensible set of fine-grained attributes of the packet, such as the IP source address and the IP TTL (hop count). In particular, this parameter receives the actual fields that matched wildcards in the filter. It may be considered to be a structured binary encoding of an addressSpec that fully matches the received packet.

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

```
void     sendPacket(NetBuffer msg, AddressNet dest,
                    Attrib attrib) throws IOException;
```

The arguments to `send` perform analogous functions.

This design allows ASP AAs to be completely event-driven.

To implement channels, the ASP EE creates a channel thread per AA (i.e., per ASP process). This thread performs the upcalls to `receivePacket`. The single thread preserves ordering of packets, and ensures that there cannot be more than one packet-reception upcall at the same time.

4.2 Network Interfaces

A *network interface* is the attachment point for a node to the network. Originally each network interface on a node corresponded to a particular hardware interconnection device. In modern network-connected operating systems, 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). As a result, the OS is likely to know about many more logical/virtual network interfaces than there are physical network interface hardware devices.

The ASP EE builds a single internal list of network interfaces for all network I/O, both native IP and virtual. This master interface table thus includes the logical, virtual, and physical interfaces known to the real OS, for use with native IP connectivity, as well as the virtual interfaces created by VNET. This table is built dynamically from boot-time configuration information and from subsequent PPI calls to add or delete network interfaces.

The ordinal position of an entry in this master interface table is known as the *logical interface number* or LIN. Every defined interface has a unique LIN value, but when an interface is deleted, its LIN is not reused. Thus, the space of LIN values at any time is a set of integers, forming a series that is contiguous with gaps and perhaps quite sparse. (This design has the theoretical problem that the ASP EE cannot in general run forever)

There are PPI calls that allow an AA to find a particular interface object in the master table or to get a copy of the complete table.

An AA can explicitly open a channel for a particular network interface by including the LIN number within the `protocolSpec`:

```
/vif<LIN>/vn/vt/asp or /if<LIN>/ipv4/udp/asp
```

When an AA receives a packet from the network, the receive attributes set by the `receivePacket()` call include the LIN number for the network interface on which the packet arrived. When an AA sends a packet to the network, it can set the LIN value in the send attributes parameter to the

`sendPacket()` call, to determine which interface is used to send the packet. This information about, and control over, interfaces may not be complete, depending upon particulars of the OS in which the ASP EE is running.

4.3 VNET: Virtual Network Connectivity

VNET is a package within the ASP EE that provides a general object-oriented framework for the implementation of layered protocols. While it implements a useful stack, its implementation currently has significant limitations of function. It supports only virtual point-point links, not virtual multiaccess interfaces. It support unicast but not multicast. It does not support fragmentation or reassembly at the virtual internetwork layer. Its network-layer address structure is completely flat, providing only host addresses with no network numbers.

The rest of this section describes VNET in some detail. The formats of all VNET packet headers are presented in Appendix B.2.

4.3.1 Introduction

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. The `address` class is used to name instances of an entity at that layer. Finally, the `header` class defines the representation of the protocol header in a packet.

Packets traverse a protocol stack by VNET mapping the appropriate local address – source or destination, for sending or receiving – for a given protocol layer to a `layer` object instance with that address. The packet is then handed to that `layer` instance for subsequent processing. The VNET protocol stacks on a given node will have `layer` instances corresponding to the 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.

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 implementation must designate a `layer` instance as the "default" to handle all packet requests for unknown address values.

Layered above the protocol stacks and below the channel code there is an internal interface that mimic the Java JDK socket interfaces DatagramSocket, ServerSocket, and Socket, to provide roughly analogous services. In the Java JDK, DatagramSocket is used as the API to UDP/IP, whereas ServerSocket and Socket are used as the server and client interfaces to TCP/IP connections, respectively.

In addition, VNET provides a network management interface to create, modify, and query the network interface and routing tables used by the VNET protocols.

4.3.2 Datagram API

The unreliable datagram socket interface of VNET mimics the UDP socket interface in the Java JDK. We merely highlight the subtle differences rather than providing full documentation for the VNET API.

- VNET defines a datagram socket class SocketDatagramV and a datagram packet class DatagramV, corresponding respectively to the Java JDK DatagramSocket and DatagramPacket classes.
- VNET uses classes AddressVN for network layer addresses and AddressVT (32-bit integers) for transport layer ports, corresponding respectively to the Java JDK class InetAddress and primitive type int used for ports.

VNET datagram sockets provide two features not found in the Java JDK.

- Zero-Copy Interface

  Alternative send and receive methods provide the ability to send a receive a NetBuffer object and avoid the data copy which occurs with the use of the standard DatagramPacket object. When performing a send, the application passes down a NetBuffer to the protocol stack. This buffer should contain empty space at the top of the buffer which can be used by VNET to insert the appropriate protocol headers. Applications must assume that the transmission of the packet will be asynchronous with respect to the send method invocation. This implies that no further references should be made to the NetBuffer once it has been passed to VNET.

- Hop-by-Hop Processing

  Applications can send datagrams using a DatagramSocketVS if they want hop-by-hop interception of the datagram along a path. Such datagrams are addressed to a final destination but never travel more that one hop towards that destination. If there is no open socket at the intermediate router, the datagram is dropped rather than forwarded. This is analogous to the use of the IP Router Alert option or the RSVP proto 46 intercept mechanisms.

  The port space of the DatagramSocketVS is distinct from the DatagramSocketV sockets. Therefore, an application needs an open DatagramSocketVS socket on the appropriate port at each router along the path.

4.3.3 Reliable Stream API

To provide a reliable byte stream analogous to the Java JDK's TCP-based socket interface, VNET provides an interface to a Java implementation of the Reliable Data Protocol (RDP) [5]. 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.

A description of the properties of RDP and an overview of its implementation in Java are found in [2].

- VNET defines the client and server socket classes ServerSocketV and SocketV, corresponding respectively to the Java JDK classes ServerSocket (passive open) and Socket (active open).
- VNET uses classes AddressVN for network layer addresses and AddressRDP (32-bit integers) for RDP ports, corresponding respectively to the Java JDK class InetAddress and primitive type int used for ports.

4.3.4 The Network Management Interface

The network management interface provides the ability to add, delete, change, or query elements of the network interface and routing tables. Applications can also register for upcalls from the NMI informing the application that something in these tables has been changed by another application.

4.3.5 Layer 4 Protocol Implementations

This section describes the Layer 4 protocol implementations that are available in the current prototype. Each of these layers is a subclass of the LayerL4 abstract base class.

- LayerVT – UDP-Like Datagram Service

  This layer 4 protocol implementation provides a datagram service analogous to UDP. This layer uses VNET source and destination ports chosen from a VNET port space of 32-bit integers. Only one layer instance can be bound to a particular local VNET port at any time. Incoming datagrams are queued internally for deliver to applications, and packets are dropped if these queues are filled to capacity.

- LayerVTS – UDP-like Datagram Service with H/H Processing

  This layer provides the same service as LayerVT but with hop-by-hop service. The port space defined in this layer is distinct from the port space used by LayerVT. Thus, it is possible to have layers bound to the same port number in each of these spaces simultaneously.

- LayerRDP

  This layer provides reliable datagram delivery across VNET (or UDP). For details on the protocol see [5].

4.3.6 Internetwork (Layer 3) Protocol Implementations

This section describes the Layer 3 protocol implementations that are available in the current prototype. Each of these layers is a subclass of the LayerL3 abstract base class.

- LayerVN – Internetwork Layer

  The LayerVN implementation provides a network layer similar to IP but with fewer features. Packets are forwarded end-to-end using only point-to-point interfaces. The source and destination addresses in this layer are VNET internetwork addresses, which use a completely flat space (no host/network distinction) of 32-bit integers.

  No packet fragmentation or reassembly is supported. Packets carry a time-to-live (TTL) field to prevent excessive resource consumption in the presence of routing loops.

- LayerVNS – Internetwork Layer with H/H Processing

  The LayerVNS network layer shares interfaces and routes with the LayerVN layer. The main difference between these two layers is that there is no forwarding engine in LayerVNS, since packets are forwarded only one hop before they are captured for processing by a AA. When a packet arrives, it is either delivered to the corresponding layer 4 protocol stack or it is dropped. Because packets are never forwarded, a TTL field is unnecessary for this protocol.

4.3.7 Link Layer (Level 2) Protocol Implementations

This section describes the Layer 2 protocol implementations that are available in the current prototype. Each of these layers is a subclass of the LayerL2 abstract base class.

- LayerUI – Virtual Link Layer using UDP/IP Encapsulation

  LayerUI packets are framed using a a HeaderUI header as well as UDP and IP headers. The packet size at this layer is limited to the IP message size limit of 64KB less the header overhead introduced by LayerUI.

  Addressing at this layer uses AddressUI objects as virtual link layer addresses; these contain an IP address and a UDP port number. The external format of a HeaderUI Object is shown in B.2.

  An instance of a LayerUI object is created by supplying a local AddressUI for that instance. The LayerUI implementation opens a socket on the given local IP address and UDP port number. When packets are passed down from Layer 3, they are encapsulated with a HeaderUI header and sent on this socket to the appropriate destination. A listening thread receives datagrams from this socket, decapsulates the packet, and sends it to the appropriate Layer 3 protocol for subsequent processing.

- LayerAnep – Virtual Link Layer using ANEP/UDP/IP Encapsulation

  LayerAnep implements a VNET virtual link layer with ANEP demultiplexing and UDP/IP encapsulation. Packets are framed using both a UDP/IP header and an ANEP header, but no VNET-specific header. The packet size at this layer is limited to the IP message size limit of 64KB less the header overhead introduced by the ANEP header.

  Virtual link layer addresses for this layer use AddressAnep objects, which contain an IP address, UDP port number, and the ANEP Type ID.

An instance of a LayerAnep object is created by supplying a local AddressAnep for that instance. The LayerAnep implementation opens a socket on a given local IP address and UDP port number. When packets are passed down from Layer 3 of VNET, they are encapsulated with an ANEP header and sent on this socket to the appropriate destination. A listening thread receives datagrams from this socket, decapsulates the packet, and sends it to the appropriate Layer 3 protocol for subsequent processing.

The LayerAnep class has been designed to operate with both Java JDK UDP sockets and Anetd [7]. Packets arriving from Anetd appear on the standard input to the JVM rather than a particular UDP port number. A LayerAnep object instance can be created using port 0 to listen on standard input. All other non-zero port numbers cause LayerAnep to listen on the corresponding UDP port number.

A AAspec Format

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

```
<AAspec>  ::= <Q-AAname> [ <space> <Q-searchPath> <space> <Q-AAbaseName> ]
                  { <space> <Q-SearchPath> <space> <Q-ClassName-List> }

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

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

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

<Q-AAbaseName>     ::= " <className> "

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

<IPaddr>           ::= <hostname or numeric IPv4/IPv6 address>
<VNETAddr>         ::= <32-bit integer>
<className>        ::= <Fully-qualified Java class name>
<URL>              ::= <Uniform Resource Locator>
<AAname>           ::= <AAnameChar> { <AAnameChar> }
<AAnameChar>       ::= <letter> | <digit> | ! | # | $ | % | & | = | + | - | _ | @
```

It may be seen that the AAspec consists of a set of qouted strings (denoted by "Q-" prefixes). It is generally case-sensitive.

The AAname is assumed to be globally unique. `<SearchPath>` contains a list of code-servers from which 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 `[ <Q-searchPath> <Q-AAbaseName> ]` pair is optional for AAspecs contained in packets, but the pair is required for AAspecs in the `AAspec.conf` file.

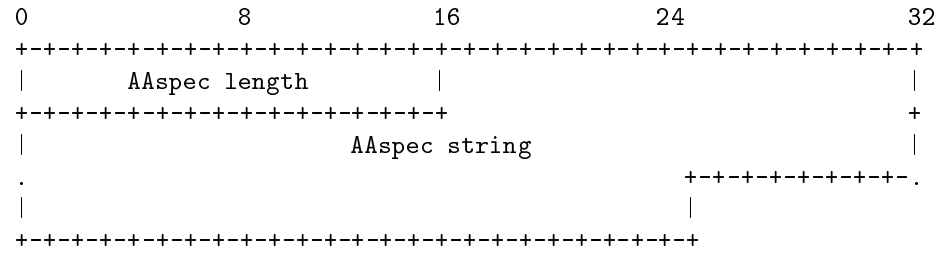
`<Q-AAbaseName>` contains the name of the primordial *AAbase* class, which must extend `AAcontext`. The EE loads and instantiates the *AAbase* class when the corresponding AAname is first requested. The EE saves a reference to this instance and maintains a one-to-one correspondence between AAspec and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

The optional pair `[ <searchPath> <listOfClassNames> ]` can be used to define distinct search paths for a specified list of Java classes. This feature allows developers to extend an AA using extension classes that reside in a code-server different from the base AA classes.

The following string is an example of an AAspec for an RSVP AA. Note that the newlines are for presentation purposes only, as AAspecs contain no newlines.

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

When the AAspec appears explicitly as the header of a packet, it must have the form of a *framed AAspec*, in which the arbitrary-length AAspec string is preceded by a 2-byte length field. The length field contains the total number of bytes in the *framed AAspec* (including the length field).



B Protocol Formats

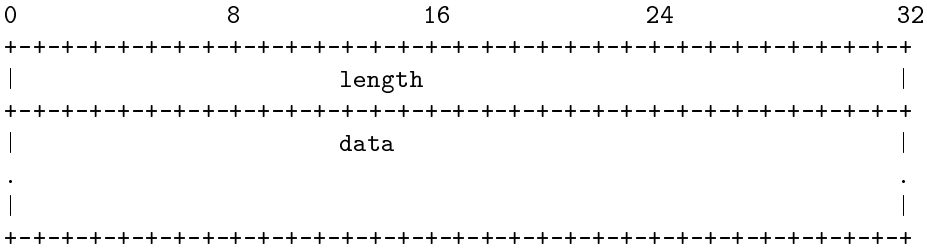
B.1 UA API Framing

Data is framed into packets on the UA to AA protocol. This framing is quite simple. Uninterpreted data is framed by a 4 byte length header at the front of the packet. The length field does not include the length of the header itself.

There are two distinct framing conventions used between an AA and its UA. These correspond directly to the standard packet demultiplexing rules of ASP which are based on implicit or explicit AAspecs.

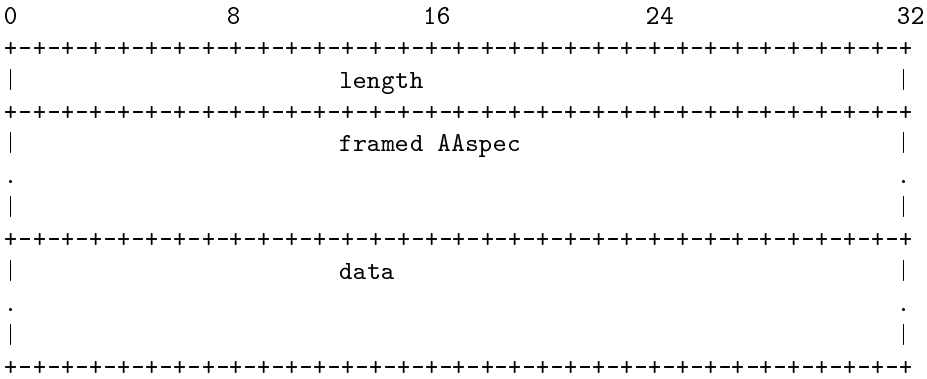
- Implicit AAspec Framing Convention

  When an InChannel is configured to invoke a given AA, the AAspec is consider to be implicitly defined and does not need to appear in the payload of the message.



- Explicit AAspec Framing Convention

  When an InChannel is configured to invoke the wildcard AA, an explicit AAspec must appear in the message.



  The AAspec determines which AA will process the request; the AA will be dynamically loaded by the EE if necessary.

The format of a framed AAspec is defined above. It consists of a 2-byte length field followed by the AAspec as an ASCII string. The class `asp.AAspec` includes the `getByteData` method which produces this format.

B.2 VNET Header Formats

B.2.1 Transport Layer Header

- HeaderVT – UDP-like Datagram Service

  The source and destination addresses in this layer are VNET port numbers, 32-bit integers.

  ```
  HeaderVT Object: Header Type = 105, Address Types = 5

  +-------------+-------------+-------------+-------------+
  |         Header Type       |      Src Address Type     |
  +-------------+-------------+-------------+-------------+
  |                     Src Address                       |
  +-------------+-------------+-------------+-------------+
  |      Dst Address Type     |     Dst Address (MSW)     |
  +-------------+-------------+-------------+-------------+
  |       Dst Address (LSW)   |
  +-------------+-------------+
  ```

- HeaderVTS – UDP-like Datagram Service with Hop-by-hop Processing

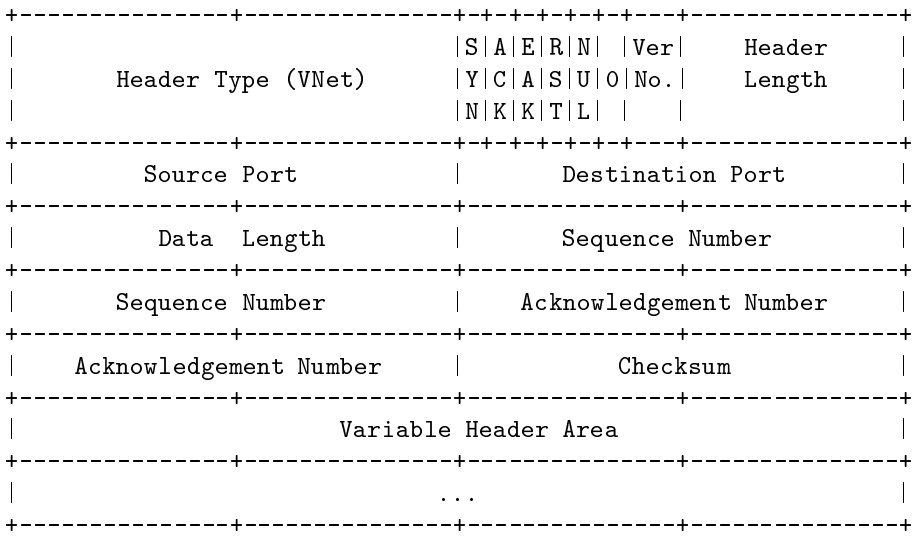
  The port space defined in this layer is distinct from the port space used by LayerVT. The source and destination addresses in this layer are VNET port numbers, 32-bit integers.

  ```
  HeaderVTS Object: Header Type = 104, Address Types = 4

  +-------------+-------------+-------------+-------------+
  |         Header Type       |      Src Address Type     |
  +-------------+-------------+-------------+-------------+
  |                     Src Address                       |
  +-------------+-------------+-------------+-------------+
  |      Dst Address Type     |     Dst Address (MSW)     |
  +-------------+-------------+-------------+-------------+
  |       Dst Address (LSW)   |
  +-------------+-------------+
  ```

- LayerRDP – RDP (Reliable Data Protocol)

  The basic layout of the RDP packets is:



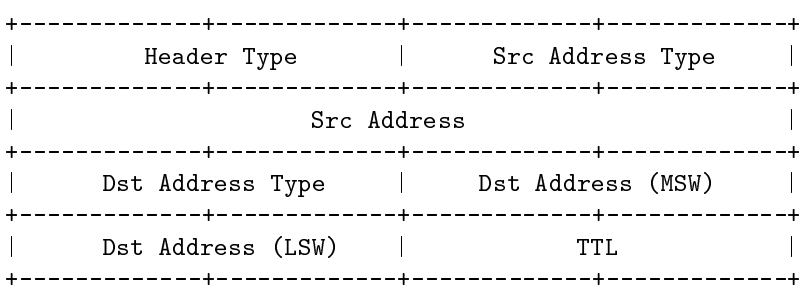
  Other than the header type, which is added for VNET, the packet format is as described in [5].

B.2.2 Internetwork (Layer 3) Protocol Headers

This section describes the VNET packet headers used for the (inter)network level or layer 3 protocol. The source and destination addresses are VNET internetwork addresses.

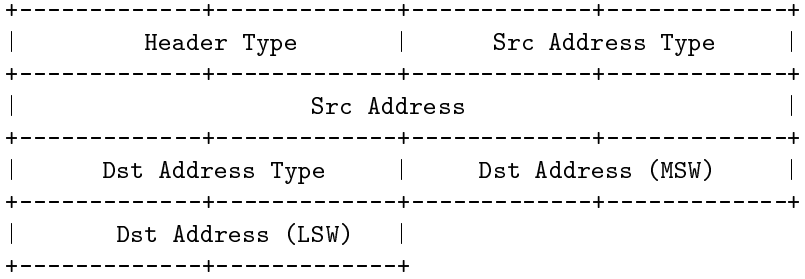
- HeaderVN – Internetwork Layer

  HeaderVN Object: Header Type = 103, Address Types = 3



- HeaderVNS – Internetwork Layer with Hop-by-hop Processing

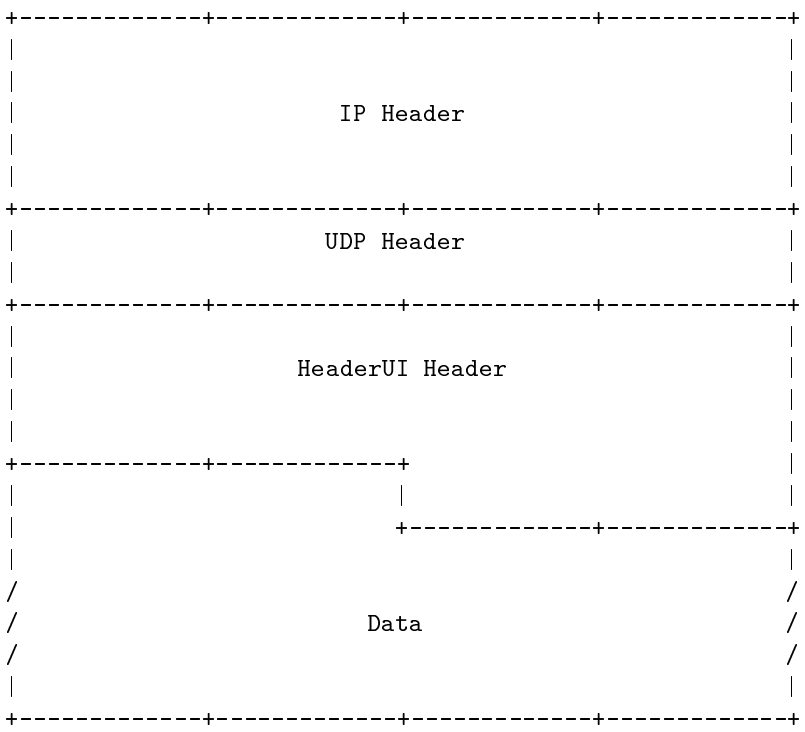
  HeaderVNS Object: Header Type = 102, Address Types = 2



B.2.3 Link Layer (Layer 2) Protocol Headers

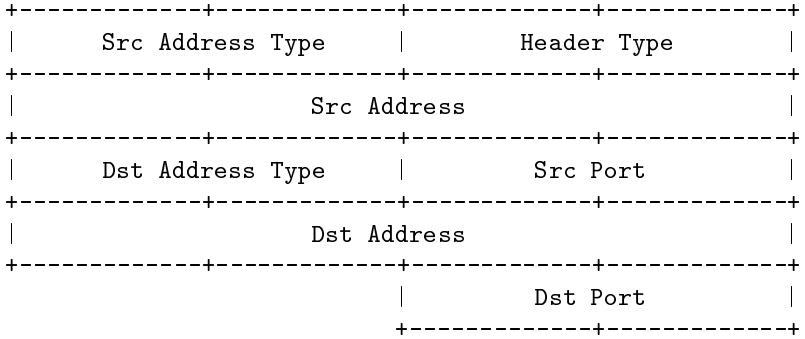
- LayerUI – UDP/IP Encapsulation

  Here a virtual link layer address is the pair (IP address, UDP port). A LayerUI packet has the following format:



  Here the HeaderUI Object format is:

```
HeaderUI Object: Header Type = 100, Address Types = 0
```



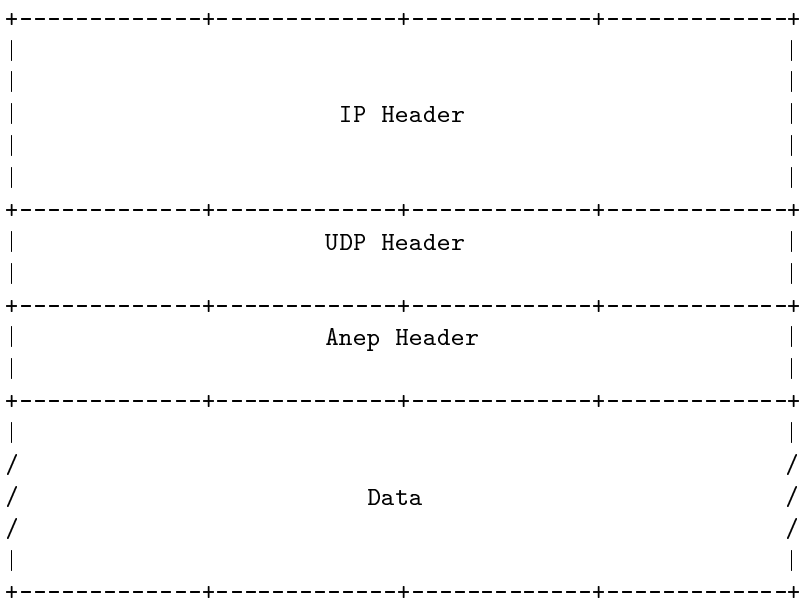
An instance of a LayerUI object is created by supplying a source AddressUI for that instance. The LayerUI implementation opens a socket on the given source IP address and UDP port number. When packets are passed down from Layer 3, they are encapsulated with a HeaderUI header and sent on this socket to the appropriate destination. A listening thread receives datagrams from this socket, decapsulates the packet, and sends it to the appropriate Layer 3 protocol for subsequent processing.

- LayerAnep – ANEP/UDP/IP Encapsulation

  LayerAnep implements a VNET link layer with ANEP demultiplexing and UDP/IP encapsulation. The packet size at this layer is limited to the IP message size limit of 64k less the header overhead introduced by LayerAnep. Addressing at this layer uses the triplet:

  ```
  (IP address, UDP port number, type ID)
  ```

  A LayerAnep packet has the following format:



  Note that there is no VNET link layer header in this case.

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

[3] Active Network Working Group. Architectural framework for active networks. http://www.cc.gatech.edu/projects/canes/papers/arch-1-0.ps.gz, July 1999.

[4] AN Node OS Working Group. Nodeos interface specification. http://www.cs.princeton.edu/nsg/papers/nodeos.ps, January 2000.

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

[6] Graham Phillips, Bob Braden, Jeff Kann, and Bob Lindell. *Writing an Active Application for the ASP Execution Environment*, 2000. http://www.isi.edu/active-signal/ARP.

[7] Livio Ricciuli. *Active Networks Daemon Protocol*, 1999. http://www.isi.edu/abone/DOCUMENTS/anetd.protocol.txt.