Writing an Active Application for the ASP Execution Environment – Release 1.6

Active Reservation Protocols (ARP) Project Staff
USC/Information Science Institute
isi-arp@isi.edu

February 18, 2003

Contents

1 Introduction

The ASP Execution Environment (ASP EE) is an active network environment for executing network protocol code written in Java as active applications (AAs). Active computations are assumed to be launched when User Applications (UAs) contact the local ASP EE using a UA/AA API. The general active networking model, including the roles of EEs, AAs, and UAs, is summarized in "Introduction to the ASP EE" [3], with which we assume some familiarity.

This document describes the programming interfaces and conventions that are needed to prepare an AA to execute under the ASP EE, including the ASP EE's interface to AAs known as the *protocol programming interface* (PPI). The Appendices describe the ASP configuration files.

1.1 ASP EE Overview

In overview, the ASP EE is planned to support:

1. Dynamic loading of AA classes.
2. Security and Resource Protection
3. Network I/O (Section 5), including support for both virtual connectivity and native IP connectivity [2].
4. The UA/AA API (Section 6).
5. Inter AA communication (Section 7).
6. A process model for execution and isolation of AAs (Section 3).
7. A soft-state repository and timing mechanism (Section 4).
8. An interface to query unicast and multicast routing (Section 8).
9. Under native IP connectivity, an interface to kernel traffic control, including packet classification and scheduling (Section 9).

Currently, there are limitations in the support for items 2, 3, 8, and 9 as described in "Introduction to the ASP EE" [3].

1.2 Coding Rules for an ASP AA

ASP AA code must obey the following rules.

1. An AA must use no *static* fields; they must use AA-local data instead. See Section 3.2.
2. No static initializer blocks are allowed. A new construct for static initializer blocks is defined in this implementation as a replacement. See Section 3.2.
3. 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.
4. 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.
5. An AA may create its own thread(s), but it must use the ASP classes `AspThread` or `AspThreadGroup` rather than `java.lang.Thread` or `java.lang.Threadgroup`, respectively.
6. Every AA must include a primordial *AAbase* class that extends the `AAcontext` class. The name of the *AAbase* class appears in the AAspec in each protocol packet. If the *AAbase* class for the requested AA is not already loaded, the ASP EE loads and instantiates the requested class.

   The *AAbase* class provides an interface through which the EE can invoke upcalls and the AA can invoke downcalls. It may also contain some of the AA-local context, primary reference fields from which the AA can locate the rest of its AA-specific data. Such primary reference data can alternatively be stored in the AA-local data space.
7. The *AAbase* class must implement the the receive upcall method `receivePacket()` from the abstract `AAcontext` class. See Section 5.
8. Every application thread should periodically call the `isAlive` function (Section 3) and exit if it returns `false`.
9. 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 Section 2 for coding examples.

Note that some of these rules imply some performance penalty over "normal" Java implementation techniques.

2 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. Then the call -

```
//nameMapper is a object of java.util.Hashtable
String CCversed_name = (String)nameMapper.get("CC");

Class c1 = AspSystem.forName(CCversed_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 `forName` 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.

This technique 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. Its drawback is that it does not allow class version binding; a particular definite class must be selected for each apparent class in each AA at compile time. This approach uses indirect constructor methods, which we call *proxy constructors*, for each dynamically bound class construction. By convention, all of these constructor proxies are gathered into the *AAbase* object.

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

```
class AAbase_AA_1 extends AAcontext {
        ...
```

```
        // Define constructor proxy methods
        //
        CC newCC(A a, B b, C c) {
                Class C1 = AspSystem.forName("CC-1-*");
                return C1.newInstance();
        }
        ...
}
```

If no version resolution is allowed, then the proxy constructor becomes even simpler.

```
class AAbase_AA_2 extends AAcontext {
        ...

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

In either case, AA classes would use the following code to construct a `CC` using dynamic binding and proxy constructors:

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

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

3 Process Management Interface

3.1 The Process Model

The EE models the execution and state of each AA as a distinct *process*.

- There is a one-to-one correspondence between ASP processes and AA instances active in a node. A process is created for an AA by the ASP EE when the AA is first loaded. A process persists until its AA is deleted.
- When a packet arrives for an EE, an EE thread is used to execute an upcall to the `receivePacket` method of the *AAbase* class for the AA. For each AA, a single thread shepherds packets across the PPI boundary, delivering them using the `receivePacket` upcall. If the thread does not return from the upcall, no additional upcalls will occur for that AA, since ASP only allows a

single outstanding `receivePacket` upcall. A single outstanding upcall is a simple mechanism to ensure that packets are upcalled to an application in received order.

- Multiple Java threads may execute within a single process. An AA may also fork its own internal, or *AA-local*, threads, which become associated with the AA process. The AA must use the `AspThread` class rather than Java's `Thread` class for this purpose.
- All execution of AA code by the ASP EE is performed under some ASP thread of the corresponding AA process.
- Processes are isolated from each other; that is, the ASP process is the unit of protection and isolation.
- The ASP EE supports a mechanism for saving data that is private to a process. This mechanism, *AA-local data* or *AA-LD*, substitutes for static fields and static initialization in AA classes. In particular, an AA process can use AA-LD to store a reference to its *AAbase* object.

The *AAbase* class inherits the following process-management methods from the `AAcontext` class:

```
String           getProcessName()

static void      putLD(String name, Object value)

static Object    getLD(String name)

static AspThread currentThread()

boolean          isAlive()

void             terminate()
```

- The `getProcessName` method returns the name of the current process. Process names correspond to the name of the AA specified in the AAspec.
- The `putLD` method places an object into AA-LD with the key `name`. 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.
- The `getLD` method retrieves an object from AA-LD with the key `name`. 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.
- Java currently limits the ASP EE from forceably terminating application threads. When an ASP process is terminated, it relies on the AA to cooperate and terminate it's outstanding

threads. Threads which are not terminated will be demoted to a minimal scheduling priority. Termination is communicated to application threads using the method `isAlive`. `isAlive` should be periodically checked by each application thread. If the function returns false, the thread should exit. Calling the method `terminate` sends this termination signal to the other threads.

3.2 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.getLD("str"));
       }
       // Assignment to "str"
       private static void str(String x) {
           AspProcess.putLD("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.putLD("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.getLD("lock");
        synchronized (lock) {
         ...
        }
    }
}
```

4 Soft-state Container

The soft-state repository mechanism within the ASP EE allows each AA to maintain multiple private tuple spaces. The mechanism supports both soft state and hard state. The soft-state support includes a timer thread that invokes upcalls to the protocol code at appropriate times.

4.1 Tuple Spaces

Each AA can instantiate multiple tuple spaces or *state containers.* The ASP EE partitions state containers among AA processes, so that state placed into a state container owned by one AA cannot be referenced by any other AA.

A state container is a repository for a set of 4-tuples:

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

- *Key* is a key for retrieving the tuple.
- *Value* is the object being stored.
- *RefreshT* is the refresh timeout interval for the tuple, in milliseconds. When this interval expires, the state container will make an `actionAtRefresh` upcall to the AA. If this value is negative, no refresh upcall will be made.

  Typically, this upcall will cause the AA to forward a refresh message to another node based upon the state in *Value*, and then call the `put` method (see below) to reset *RefreshT* for the next refresh interval.
- *TimeoutT* is the timeout interval for the tuple in milliseconds. When this interval expires, the state container will remove the tuple and then make an `actionAtTimeout` upcall to the AA. If this value is negative, no timeout upcall will be made; this will be the case for hard-state. If the value is zero, the timer value is unchanged, and the remaining time from the original timer will be used as the new timeout interval.

4.2 State Container Methods

1. Create/Find State Container

```
static StateContainer getContainer(Object containerID,
                                   ContainerAdapter adapter)
```

Creates a new state container named by the `containerID` parameter, with the helper object `ContainerAdapter` (see below). Should a state container named by `containerID` already exist, a reference to it is returned and the `ContainerAdapter` parameter is ignored. If the `containerID` parameter is a `String`, there will be a distinct container for each unique character string within each process.

The `ContainerAdapter` parameter references an instance of a helper object that defines the upcall methods for the *RefreshT* and *TimeoutT* events:

```
void actionAtRefresh(Object object)
void actionAtTimeout(Object object)
```

Should the AA not need any upcall processing, the AA can implement these methods with a single `return` statement.

2. Add/Modify Tuple in Container

```
Object          put(Object key, Object value, int timeoutI,
                        int refreshI)
```

If a tuple is found whose *Key* component (see below) matches `key`, the corresponding *Value* component is replaced by `value`, the two timer values are updated from `timeoutI` and `refreshI`, and the original *Value* component is returned. If no matching tuple is found, a new tuple is inserted into the state container and `null` is returned.

3. Find Value from Container

```
Object          get(Object key)
```

This method returns a reference to the *Value* object of the tuple selected by `key`, or it returns `null`.

4. Enumerate all Keys

```
Enumeration     keys()
```

This method returns an enumeration containing all the keys in the container. The caller can then search these keys in an arbitrary manner. Note that this method is inefficient when the number of tuples is large because it iterates through the entire container.

5. Remove a Tuple

```
Object          remove(Object key)
```

This method explicitly removes the tuple named by `key` from the state container, if one exists.

6. Read Container Size

```
int             size()

boolean         isEmpty()
```

The `size()` method returns the number of unique tuples in the container, while the method `isEmpty` returns `true` iff the container is empty.

4.3 State Container Keys

The AA must define a `key` class for locating values stored in a state container. A `key` class will generally define one or more fields used to select a tuple, as well as two methods that define the selection algorithm: `equals` and `hashCode`.

```
public boolean equals(Object o)

public int hashCode()
```

Because of these two methods, a `key` parameter is active, giving considerable flexibility and an opportunity for search optimization. The state container access rules work as follows.

- The `StateContainer` uses a closed hash table.
- The `hashCode` method is used to find the index of a sublist vector; this sublist is then searched until the `equals` method returns `true`. Note that the *Key* components of the tuples are compared, not the *Value*s.

The `hashCode` method in the `put` operation precisely controls how tuples are partitioned into sublists. During a `get` or `put` operation, the `hashCode` and `equals` methods together determine which value is retrieved.

There is no requirement that the `Key` objects be the same class during `put` and `get` operations; the active keys can be tailored to control the search algorithms. This flexibility allows the AA to efficiently take certain slices through a multi-dimensional key space, and to perform particular operations such as efficiently finding objects in a sublist that don't conform to a given property.

For example, suppose the key for a tuple contains the value pair (X, Y). If `hashCode` depends upon X alone, then slices (X,*) appear in the same sublist, which may allow relatively efficient searches of such slices. To check that a sublist satisfies some property, a `key` can be used whose `equals` method returns `true` if the required property does *not* hold. The `get` method will then return the first object in the sublist for which the property fails (because this will be the first object whose `equal` method returns `true`). If all objects in the sublist conform, then the `get` method returns `null`. Note that the property can be defined *after* the objects are placed into the container.

5 Network I/O

The ASP EE's PPI provides an integrated interface to two alternative network protocol stacks: a virtual network stack called *VNET*, and a native IP stack (see [3]). This network I/O interface is based upon the *channel abstraction* of the node OS interface [5], which defines InChannels for receiving packets and OutChannels for sending packets. Previous EE versions used a different interface that is no longer supported.

5.1 I/O Parameters

5.1.1 ProtocolSpecs

The protocolSpec strings that are currently supported by the PPI are:

- Virtual datagram transport layer: `"vif[0-9]*/vn/vt"`

  This channel performs network-layer and transport-layer processing in VNET, delivering the payload of a VNET datagram. The optional integer following `vif` specifies a (virtual) interface number, also called a Logical Interface Number or LIN. The corresponding `addressSpec` defines demux values in the VNET (virtual) spaces.

- Virtual datagram EE layer: `"vif[0-9]*/vn/vt/asp"`

  This channel extends the "vif[0-9]*/vn/vt" protocolSpec by adding the the EE's header on output or stripping it on input. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Virtual reliable transport layer: `"vif[0-9]*/vn/rdp"`

  This channel performs network-layer and transport-layer processing in VNET, delivering a reliable byte stream. The optional integer following `vif` specifies a (virtual) interface number, also called a Logical Interface Number or LIN.

- Virtual reliable EE layer: `"vif[0-9]*/vn/rdp/asp"`

  This channel extends the "vif[0-9]*/vn/rdp" protocolSpec by exchanging the EE's header only once with the remote end when the RDP connection is established. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Virtual divert channel: `"vif"`

  This channel diverts all VNET datagrams that match a given `DemultiplexKey`, regardless of whether the EE is the destination host for the datagram or not. The channel does not do any network-layer processing and delivers the raw VNET datagram (ie. including the network and transport headers). Support for opening such a channel on a specific LIN (eg. `"vif[0-9]*"`) is not currently supported.

- Native IPv4 datagram transport layer (UDP): `"if[0-9]*/ipv4/udp"`

  This channel performs network-layer and transport-layer processing in the native IP stack, delivering the payload of a UDP datagram. The corresponding `addressSpec` defines demux values in the native IP space.

- Native IPv4 datagram EE layer: `"if[0-9]*/ipv4/udp/asp"`

  This channel extends the "if[0-9]*/ipv4/udp" protocolSpec by adding the the EE's header on output or stripping it on input. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Native IPv6 datagram transport layer (UDP): `"if[0-9]*/ipv6/udp"`

  This channel performs network-layer and transport-layer processing in the native IPv6 stack, delivering the payload of a UDP datagram. The corresponding `addressSpec` defines demux values in the native IP space. It is currently in the experimental stage and not fully tested.

- Native IPv6 datagram EE layer: `"if[0-9]*/ipv6/udp/asp"`

  This channel extends the "if[0-9]*/ipv6/udp" protocolSpec by adding the the EE's header on output or stripping it on input. This may result in loading and executing of an arbitrary new AA by the ASP EE. It is currently in the experimental stage and not fully tested.

- Native IP stream transport layer (TCP): `"if[0-9]*/ipv4/tcp"`

  This channel performs network-layer and transport-layer processing in the native IP stack, delivering a reliable byte stream. The corresponding `addressSpec` defines demux values in the native IP space.

- Native IP stream EE layer: `"if[0-9]*/ipv4/tcp/asp"`

  This channel extends the "if[0-9]*/ipv4/tcp" protocolSpec by exchanging the EE's header only once with the remote end when the TCP connection is established. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Native IP reliable datagram transport layer (RDP): `"if[0-9]*/ipv4/udp/rdp"`

  This tunnels the reliable datagram protocol across UDP (actually across the VNet LayerUI). The corresponding `addressSpec` defines demux values in the native IP space.

- Native IP reliable datagram EE layer: `"if[0-9]*/ipv4/udp/rdp/asp"`

  This channel extends the "if[0-9]*/ipv4/udp/rdp" protocolSpec by exchanging the EE's header only once with the remote end when the RDP connection is established. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Native IPv4 network layer: `"if[0-9]*/ipv4"`

  This channel performs network-layer processing in the native IP stack, delivering as payload any transport protocol data built on top of it. The corresponding `addressSpec` defines demux values in the native IP space.

- Native IPv4 EE layer: `"if[0-9]*/ipv4/asp"`

  This channel extends the "if[0-9]*/ipv4" protocolSpec by adding the the EE's header on output or stripping it on input. This may result in loading and executing of an arbitrary new AA by the ASP EE.

- Native IPv4 network layer with IP header: `"if[0-9]*/ipv4h"`

  This channel performs network-layer processing in the native IPv4 stack, delivering as payload any transport protocol data built on top of it as well as the native IPv4 header of the packet. The corresponding `addressSpec` defines demux values in the native IP space.

- Native IPv6 network layer: `"if[0-9]*/ipv6"`

  This channel performs network-layer processing in the native IPv6 stack, delivering as payload any transport protocol data built on top of it. The corresponding `addressSpec` defines demux values in the native IP space. It is currently in the experimental stage and not fully tested.

- Native IPv6 EE layer: `"if[0-9]*/ipv6/asp"`

  This channel extends the "if[0-9]*/ipv6" protocolSpec by adding the the EE's header on output or stripping it on input. This may result in loading and executing of an arbitrary new AA by the ASP EE. It is currently in the experimental stage and not fully tested.

- Native IPv6 network layer with IP header: `"if[0-9]*/ipv6h"`

  This channel performs network-layer processing in the native IPv6 stack, delivering as payload any transport protocol data built on top of it as well as the native IPv6 header of the packet. The corresponding `addressSpec` defines demux values in the native IP space.

- UA API: `"api"`

  This is a stream-based API InChannel. Each API channel corresponds to a local TCP connection between the ASP EE and a UA.

  No `addressSpec` parameter is needed for an API channel. API channels have several other peculiarities; see Section 6.

- Inter AA communication channel: `"ipc"`

  This provides a unidirectional channel that can be used to share information between cooperating Active Applications(AA).

  Inter AA channels have several peculiarities; see Section 7.

5.1.2 AddressSpecs

The `addressSpec` is implemented as a specialization of the `asp.net.Attrib` abstract base class, which is used as extensible parameters for I/O calls. Depending on the level in the network stack the `asp.net.Attrib` is specalised into an appropriate attribute class for that level (i.e., for network level protocols, there is a `asp.net.AttribNetwork` class implementation). The I/O attributes in these classes are represented as public fields rather than public methods. The following attributes are currently implemented, but this list may be extended in the future - `asp.net.AttribInterface`, `asp.net.AttribNetwork`, `asp.net.AttribTransport` and `asp.net.AttribIPC`.

The `asp.net.AttribIPC` implements the addressSpec for the Inter AA communication channel (*ipc*). Only an object of the AttribIPC class is accepted as a valid instance of the `addressSpec` by the ASP EE for this type of channel.

```
public class AttribIPC extends Attrib {
        public String     dst_aaspec;
        public String     chan_name;
}
```

- `dst_aaspec` is the AASpec of the reader/writer AA depending upon its usage.
- `chan_name` is the unique channel name as specified while creating the channel.

For the interface layer channels i.e. divert channel with protocolSpec *vif*; only objects of `asp.net.AttribInterface` are accepted as a valid instances of the `addressSpec` by the ASP EE.

```
public class AttribInterface extends Attrib {
        public int              LIN;
        public AddressNet       inf_addr;
 }
```

- `LIN` is the logical interface number of the incoming interface when it can be determined. Otherwise it is set to `AttribInterface.NO_LIN`. The `LIN` value can be used in channel protocol specifications to specific a particular interface.
- `inf_addr` field is used internally by the EE.

For the network layer channels i.e. channels with protocolSpec if[0-9]*/ipv4 and if[0-9]*/ipv4/asp; only objects of `asp.net.AttribNetwork` are accepted as a valid instances of the `addressSpec` by the ASP EE.

```
public class AttribNetwork extends AttribInterface {
        public short            protocol;
        public AddressNet       local_addr;
        public AddressNet       remote_addr;
        public short            ttl;
        public int              tos;
 }
```

- `protocol` is the transport protocol number.(i.e., for udp it is 17)
- `local_addr` is the network-layer address of the local node. Normally the system would supply the appropriate local address, but a node may set it to masquerade as another node.
- `remote_addr` is the ultimate destination address to which the packet was addressed.
- `ttl` is the received time-to-live value (TTL). Value zero means unknown TTL; in this case, it is best not to drop the packet. Its value will be zero for packets received via VNET with `TOS_HBH` service.

- `tos` defines the network-layer "type of service"; values are `asp.net.TOS_E2E` (end-to-end) or `asp.net.TOS_HBH` (hop-by-hop). This is currently supported only for VNET.

For the transport layer channels i.e. channels with protocolSpec vif[0-9]*/vn/vt, vif[0-9]*/vn/rdp, if[0-9]*/ipv4/udp, if[0-9]*/ipv4/udp/rdp, if[0-9]*/ipv4/tcp and the corresponding EE layer variants; only objects of `asp.net.AttribTransport` are accepted as a valid instances of the `addressSpec` by the ASP EE.

```
public class AttribTransport extends AttribNetwork {
        public int              local_port;
        public int              remote_port;
 }
```

- `local_port` is the transport-layer source port.
- `remote_port` is the transport-layer port to which a packet should be sent.

5.1.3 DemultiplexKey

In ASP the `DemultiplexKey` is implemented by `asp.net.DemultiplexKey` class, which is used to filter the incoming packets. One can create a demux key by using the public constructor method and retrieve the demux key fields by using the appropriate `get*()` methods. For example, the offset field can be retrieved using `getOffset()`.

Currently ASP supports only the generic demultiplex key format.

```
public class DemultiplexKey implements Cloneable {
        private short            operator;
        private int              offset;
        private byte[]           mask;
        private byte[]           test;

        // Valid operators for matching a byte pattern
        public static final short NONE = 0;     //Defaults to Equals
        public static final short EQ = 1;       //Equals =
        public static final short NEQ = 2;      //Not Equals !
        public static final short LT = 3;       //Less than <
        public static final short LTEQ = 4;     //Less than Equals <=
        public static final short GT = 5;       //Greater than >
        public static final short GTEQ = 6;     //Greater than Equals >=


        public    DemultiplexKey(int offset, byte[] mask,
```

```
                                        byte[] test, short operator)
}
```

- `DemultiplexKey()` is used to construct a new DemultiplexKey object
- `operator` is the operator to be applied for matching a set of bytes at a particular offset in a packet. The set of currently valid operators is listed above.
- `offset` indicates that the specified byte pattern will be compared at offset bytes from the beginning of the payload
- `mask` indicates a bit pattern that is applied to the packet at offset bytes. Only bits that are active in the mask are checked for a match.
- `test` indicates a sequence of bytes the demux key should match in a packet.

5.1.4 ChannelHandlers

A `ChannelHandler` is an interface that specifies methods for receiving packets and catching exceptions from input processing on a channel. Specifically the interface is:

```
interface ChannelHandler {
        public void receivePacket(InChannel chan,
               NetBuffer msg, Attrib attrib) throws IOException;

        public void receiveException(Channel chan, Exception e);
}
```

- `receivePacket` is called for each incoming packet on the channels handled by the object implementing it. This method is discussed in detail below.
- `receiveException` is called if an exception is raised during input processing for this channel. The parameters indicate which channel threw the exception and the exception itself. Exceptions on output are thrown directly by the channel output methods.

  One common use of the `receiveException` method is to detect that the other end of a connection-oriented channel has closed the connection. When such a channel is closed, an `EOFException` is passed to `receiveException`.

`AAContext` implements `ChannelHandler` so that the emphAABase class of an AA can always be used as the handler for a channel. `ChannelHandler`s allow AAs that open several channels to use separate objects to encapsulate the processing and state of those channels.

5.2 Network Input

5.2.1 AA Explicitly Open an InChannel

```
InChannel PPI.openInChannel(
        String protocolSpec, Attrib Rattrib,
        DemultiplexKey demuxKey,
        [, ChannelHandler handler ]) throws IOException;
```

This public static constructor creates an InChannel bound to this AA, and returns a reference to it. An `IOException` is thrown if the channel cannot be created. An `openInChannel` call will fail if it specifies the same parameters as a channel that is already open.

The parameters are:

- `protocolSpec`

  A string naming the legacy protocol layers that will be processed in the received packet. May be native IP or virtual input channel, but not the API. See Section 5.1.

- `Rattrib` and `demuxKey`

  A receive attribute object that plays the role of the node OS addressSpec, binding the demultiplexing fields necessary for the `protocolSpec` processing.

  The following table shows the fields of the `Rattrib` and `demuxKey` parameters that are *required* depending on the value of the `protocolSpec`:

protocolSpec	Rattrib	demuxKey
"vif[0-9]*/vn/vt"	local_port,tos	<ignored>
"vif[0-9]*/vn/vt/asp"	local_port,tos	<ignored>
"vif[0-9]*/vn/rdp"	local_port	<ignored>
"vif[0-9]*/vn/rdp/asp"	local_port	<ignored>
"vif"	<ignored>	offset,mask,value
"if[0-9]*/ipv4/udp"	local_port	<ignored>
"if[0-9]*/ipv4/udp/asp"	local_port	<ignored>
"if[0-9]*/ipv6/udp"	local_port	<ignored>

"if[0-9]*/ipv6/udp/asp"	local_port	<ignored>
"if[0-9]*/ipv4/udp/rdp"	local_port	<ignored>
"if[0-9]*/ipv4/udp/rdp/asp"	local_port	<ignored>
"if[0-9]*/ipv4/tcp"	local_port	<ignored>
"if[0-9]*/ipv4/tcp/asp"	local_port	<ignored>
"if[0-9]*/ipv4"	protocol, local_addr, remote_addr	offset,mask,value
"if[0-9]*/ipv4/asp"	protocol, local_addr, remote_addr	offset,mask,value
"if[0-9]*/ipv4h"	protocol, local_addr, remote_addr	offset,mask,value
"if[0-9]*/ipv6"	protocol, local_addr, remote_addr	offset,mask,value
"if[0-9]*/ipv6/asp"	protocol, local_addr, remote_addr	offset,mask,value
"if[0-9]*/ipv6h"	protocol, local_addr, remote_addr	offset,mask,value

- `handler`

  The object that will have its `receivePacket` and `receiveException` methods called when packets are received or exceptions raised on input.

5.2.2 Receive a Packet

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

All packets selected by an InChannel will be delivered to the associated AA by upcalling the `receivePacket` method in the AA's *AAbase* class or in the `ChannelHandler` specified when the channel was opened.

The parameters are:

- `chan`

  The channel through which the packet arrived.

- `msg`

  A buffer object containing the received packet.

- `Rattrib`

  A reference to an `asp.net.Attrib` object that specifies complete demuxing information and some secondary control attributes of the received packet.

  The following table shows the fields of the `Rattrib` parameter that are set by the EE depending on the value of the `protocolSpec`:

protocolSpec	Rattrib
"vif[0-9]*/vn/vt"	local_port, remote_port, local_addr, remote_addr, tos, ttl, LIN
"vif[0-9]*/vn/vt/asp"	local_port, remote_port, local_addr, remote_addr, tos, ttl, LIN
"vif[0-9]*/vn/rdp"	local_port, remote_port, remote_addr, LIN
"vif[0-9]*/vn/rdp/asp"	local_port, remote_port, remote_addr, LIN
"vif"	<none>
"if[0-9]*/ipv4/udp"	local_port, remote_port, local_addr, remote_addr, LIN
"if[0-9]*/ipv4/udp/asp"	local_port, remote_port, local_addr, remote_addr, LIN
"if[0-9]*/ipv6/udp"	local_port, remote_port,

```
                              local_addr, remote_addr,
                              LIN

"if[0-9]*/ipv6/udp/asp"       local_port, remote_port,
                              local_addr, remote_addr,
                              LIN

"if[0-9]*/ipv4/tcp"           local_port, remote_port,
                              local_addr, remote_addr,
                              LIN

"if[0-9]*/ipv4/tcp/asp"       local_port, remote_port,
                              local_addr, remote_addr,
                              LIN

"if[0-9]*/ipv4/udp/rdp"       local_port, remote_port,
                              remote_addr, LIN

"if[0-9]*/ipv4/udp/rdp/asp" local_port, remote_port,
                              remote_addr, LIN

"if[0-9]*/ipv4"               protocol, local_addr,
                              remote_addr, ttl

"if[0-9]*/ipv4/asp"           protocol, local_addr,
                              remote_addr, ttl

"if[0-9]*/ipv4h               protocol, local_addr,
                              remote_addr, ttl

"if[0-9]*/ipv6"               protocol, local_addr,
                              remote_addr, ttl

"if[0-9]*/ipv6/asp"           protocol, local_addr,
                              remote_addr, ttl

"if[0-9]*/ipv6h               protocol, local_addr,
                              remote_addr, ttl

"api"                         local_port, remote_port,
                              local_addr, remote_addr,
                              LIN

"api/asp"                     local_port, remote_port,
                              local_addr, remote_addr,
                              LIN
```

```
"ipc"                       dst_aaspec, chan_name
```

5.2.3 Close an InChannel

```
void   close()
```

If an action is attempted on a closed channel, an Exception will be thrown.

5.2.4 Additional InChannel Methods

```
String          getProtocolSpec()
Attrib          getAttrib()
DemultiplexKey  getDemuxKey()
String          toString()
ChannelHandler  getChannelHandler()
```

- `getProtocolSpec()`

  This method returns the protocolSpec that was used to construct the InChannel.

- `getAttrib()`

  This method returns the `asp.net.Attrib` object that was used to open the channel.

- `getDemuxKey()`

  This method in intended to return the `demultiplexKey` parameter that was used to open the channel.

- `toString()`

  This method returns a human-readable string encoding the internal state of the channel.

- `getChannelHandler()`

  This method returns the `ChannelHandler` used to open the channel, or the *AABase* class if no handler was given.

For the API and TCP channels, `msg.getLength()` has overloaded semantics. If `msg.getLength()=-1`, then the UA has terminated the TCP connection.

5.3 Network Output

5.3.1 AA Explicitly Open an OutChannel

```
OutChannel PPI.openOutChannel(
        String protocolSpec,
        Attrib Sattrib
        [, ChannelHandler handler ]) throws IOException;
```

This public static constructor creates an OutChannel bound to this AA, and returns a reference to it. An `IOException` is thrown if the channel cannot be created. An `openOutChannel` call will fail if it specifies the same parameters as a channel that is already open. To send a packet, an AA must explicitly open an OutChannel.

The parameters are:

- `protocolSpec`

  String that determines what protocol processing the OutChannel will perform on outgoing packets. May not be "api".

- `Sattrib`

  This send attribute object plays the role of the node OS addressSpec, binding the multiplexing fields necessary for the `protocolSpec` processing. This parameter is copied by value and therefore it can be modified and reused.

If ProtocolSpec is:	Sattrib parameters Must be set:	May be set:
"vif[0-9]*/vn/vt"	local_port,tos	remote_port
"vif[0-9]*/vn/vt/asp"	local_port,tos	remote_port
"vif[0-9]*/vn/rdp"	local_port,remote_port, remote_addr	
"vif[0-9]*/vn/rdp/asp"	local_port,remote_port, remote_addr	
"if[0-9]*/ipv4/udp"	local_port	remote_port, remote_addr
"if[0-9]*/ipv4/udp/asp"	local_port	remote_port, remote_addr
"if[0-9]*/ipv6/udp"	local_port	local_addr,

```
                                                        remote_port,
                                                        remote_addr

"if[0-9]*/ipv6/udp/asp" local_port                      local_addr,
                                                        remote_port,
                                                        remote_addr

"if[0-9]*/ipv4/tcp"     local_port,remote_port,
                        remote_addr

"if[0-9]*/ipv4/tcp/asp" local_port,remote_port,
                        remote_addr

"if[0-9]*/ipv4/udp/rdp" local_port,remote_port,
                        remote_addr

"if[0-9]*/ipv4/udp/rdp/asp" local_port,remote_port,
                        remote_addr

"if[0-9]*/ipv4"         protocol,remote_addr        local_addr,ttl

"if[0-9]*/ipv4/asp"     protocol,remote_addr        local_addr,ttl

"if[0-9]*/ipv4h"        protocol,remote_addr        local_addr,ttl

"if[0-9]*/ipv6"         protocol,remote_addr        local_addr,ttl

"if[0-9]*/ipv6/asp"     protocol,remote_addr        local_addr,ttl

"if[0-9]*/ipv6h"        protocol,remote_addr        local_addr,ttl

"ipc"                   dst_aaspec                  chan_name
```

In both UDP and VNET cases, the attribute parameter must be non-`null`. Each AA has associated with it default addressSpecs which the AA may obtain using the `PPI.getDefaultAddressSpec()` method. Currently, only default source and destination *ports* are supported. Default addressSpecs are established at EE boot time. If an AA has an channel entry in the `asp.conf` file (see A.1) then default values are obtained from this entry. Otherwise EE-wide values are used. The EE-wide port values can be set by the `-x` command-line option (and the source and destination ports will be identical).

- `handler`

  The handler for an output channel is only used to process exceptions that occur asynchronously on the channel. Such exceptions occur on channels that use connection-oriented protocols like TCP or RDP. An example of such an asynchronous exception is a connection closure.

5.3.2 Sending a Packet

```
sendPacket(NetBuffer msg, AddressNet to
            [, Attrib Sattrib ] ) throws IOException
```

The parameters are:

- `msg`

  A buffer object defining the packet to be sent.

- `to`

  The target address for the message. Ignored for an API and Inter AA channel.

- `Sattrib`

  An optional parameter defining the send attributes. If it is omitted, the channel uses those attributes established when the outChannel was opened. The `remote_port` field be set either by Sattrib or in the openOutChannel call.

  If specified here, the attributes have effect only during the `sendPacket()` method's invocation; in particular, they do not replace the internal attributes that were assigned when the OutChannel was constructed. The Sattrib parameter is ignored for an API and Inter AA OutChannel.

ProtocolSpec:	Sattrib parameters may be set:
"vif[0-9]*/vn/vt"	remote_port
"vif[0-9]*/vn/vt/asp"	remote_port
"vif[0-9]*/vn/rdp"	<ignored>
"vif[0-9]*/vn/rdp/asp"	<ignored>
"if[0-9]*/ipv4/udp"	remote_port,remote_addr
"if[0-9]*/ipv4/udp/asp"	remote_port,remote_addr
"if[0-9]*/ipv6/udp"	remote_port,remote_addr
"if[0-9]*/ipv6/udp/asp"	remote_port,remote_addr
"if[0-9]*/ipv4/tcp"	<ignored>
"if[0-9]*/ipv4/tcp/asp"	<ignored>

```
"if[0-9]*/ipv4/udp/rdp"      <ignored>

"if[0-9]*/ipv4/udp/rdp/asp"  <ignored>

"if[0-9]*/ipv4"              remote_addr,ttl

"if[0-9]*/ipv4/asp"          remote_addr,ttl

"if[0-9]*/ipv4h"             remote_addr,ttl

"if[0-9]*/ipv6"              remote_addr,ttl

"if[0-9]*/ipv6/asp"          remote_addr,ttl

"if[0-9]*/ipv6h"             remote_addr,ttl

"api"                        <ignored>

"api/asp"                    <ignored>

"ipc"                        <ignored>
```

5.3.3 Close an OutChannel

```
void   close()
```

5.3.4 Other OutChannel Methods

```
String    getProtocolSpec()
Attrib    getAttrib()
String    toString()
ChannelHandler  getChannelHandler()
```

These methods are similar to those in InChannel.

5.4 Network Interfaces

The following classes, used internally by the ASP EE for network interfaces, may be needed by an AA that needs to keep track of interfaces. Each is an extension of the `InterfaceNet` class:

- `InterfaceNet` is an abstract base class of all interfaces that may be included in the EE's internal interface table.
- `InterfaceIP` is the abstract base class for all IP interfaces. It directly extends `InterfaceNet`.
- `InterfacePhy` represents a physical interface that supports the TCP/IP stacks. It directly extends `InterfaceIP`.
- `InterfaceVir` represents a virtual interface ("vif") created by the `mrouted` routing daemon and is used ONLY for IPv4 multicast packets [8]. It may represent a real interface or a multicast tunnel. It directly extends `InterfaceIP`.
- `InterfaceVNet` is the abstract base class for all VNet interfaces. It directly extends `InterfaceNet`.
- `InterfaceL3` is the abstract base class for the VNet Layer3 interface. It directly extends `InterfaceVNet`.
- `InterfaceVN` represents a physical interface that supports the VNet network layer. It directly extends `InterfaceL3`.

In addition to the send method discussed earlier, interface classes generally support the following operations of interest to AAs. These are all public, although the internal data of the EE's interface objects has package scope and is therefore invisible to an AA.

- Get Interface Address

  ```
  AddressNet      getInfAddress()
  ```

  This method returns the network address attached to the interface. The EE ensures that each interface in the internal interface table has a unique network address.

- Get Logical Interface Number

  ```
  int             getLIN()
  ```

  The `getLIN` method returns a unique integer index, called the Logical Interface Number or LIN, for the given interface. Interfaces in the EE's internal interface table have a non-negative LIN; others have negative values. For example, API pseudo-interfaces (see Section 6) have negative LIN's and do not have entries in the EE's interface table.

- Test for Up/Down Status

  ```
  boolean         isUp()
  ```

  This method returns `true` if and only if the interface is up.

The IP interface classes support the following additional operations of interest to AAs.

- Get CIDR Prefix

```
short           getCIDRprefix()
```

  This method returns the CIDR the number of bits in the CIDR prefix for the interface address, for those network layer addresses for which a CIDR prefix is defined. For IPv4, the maximum prefix is 32 bits, while for IPv6 the maximum prefix is 128 bits.

- Get Interface Name

```
String          getName()
```

  This method returns the name that the operating system configuration associates with this logical interface.

- Get Logical Interface Number

```
int             getLINUnicast()
```

  The method `getLINUnicast` returns the same value as `getLIN`, except in the particular case of a `InterfaceVir` object; then it returns the LIN of the `InterfacePhy` that is needed to send a unicast packet out the same logical interface that is used by the `InterfaceVir` for multicast packets.

- Get MTU of Interface

```
int             getMTU()
```

  This method returns the MTU for this interface.

- Test for Multicast Capability

```
boolean         doesMC()
```

  This method returns `true` if and only if the interface is capable of multicast.

- Test for Interface Identity

```
boolean         equals(Object inf)
```

  This method tests for identity of two network interfaces. The test actually compares the network addresses, since each interface in the internal interface table has a unique network address.

- Debugging String

```
String          toString()
```

  This method returns a human-readable string describing the given interface.

A particular AA may need to maintain its own AA-specific per-interface information. This can be easily accomplished with AA-specific extension classes from the ASP network interface classes listed above.

The EE exports methods (in the `PPI` class) for finding a particular interface in the EE's internal interface table or for returning a copy (clone) of all the interfaces.

- Return Interface for a Given LIN

  ```
  InterfaceNet   QueryInterface(int LIN)
  ```

  `inf.equals( QueryInterface(inf.getLIN()) )` is true for any interface `inf` in the EE's internal interface table (i.e., for which `inf.getLIN() >= 0` holds). Note that the condition `inf == QueryInterface(inf.getLIN())` is never true because `QueryInterface()` returns a newly cloned interface.

- Return Interface with Given Address

  ```
  InterfaceNet   QueryInterface(AddressNet addr)
  ```

  Returns `null` if address cannot be found. Considers only UP interfaces.

  `addr.equals( QueryInterface(addr).getInfAddress() )` is always true if `addr` is the address of an up interface. Again, the condition `addr == QueryInterface(addr).getInfAddress()` is never true because `QueryInterface()` returns a newly cloned interface together with newly cloned fields.

- Return Physical Interface with Given Address

  ```
  Enumeration PPI.QueryInterfaces()
  int PPI.getInterfaceTableNum();
  ```

  Returns an array containing all the entries of the EE's interface table, each interface positioned at the index of its LIN. The returned array and all its elements are completely cloned so that the caller cannot access the EE's data structures.

- Add an Interface in the EE's Interface table

  ```
  void AddInterface(InterfaceNet inf)
  ```

  Provides facility to the AA for adding an VNet interface into the global interface table maintained by the EE. Currently there is no support for adding an INet interface and it throws an exception for the same.

- Delete an Interface from EE's Interface table

  ```
  void DelInterface(InterfaceNet inf)
  ```

Provides facility to the AA for deleting an VNet interface from the global interface table maintained by the EE. Currently there is no support for deleting an INet interface and it throws an exception for the same.

- Notification for changes in the EE's Interface table

  `void InterfaceChangeNotification()`

  Provides facility to the AA for receiving notifications from the EE whenever there is any changes in the set of VNet interfaces. There is no support for the AA to receive any notifications about changes in the INet interfaces.

5.5 Network Addresses

Several network address families are supported by the `*net` packages, including IPv4, IPv6 for the raw IP stack and VNET addresses for the VNET stack.

The base class for all addresses is `asp.net.AddressNet`, which includes the following operations.

```
String              getHostName(boolean DNS_lookup)
boolean             isMulticast()
boolean             isWildCard()
byte[]              toByteArray()
int                 lengthByteArray()
static AddressNet   fromByteArray(byte[] addr,int offset)
static AddressNet   getByName(String name, short family)
```

The `getHostName` method returns the hostname of the given address. If the parameter `DNS_lookup` is true then a DNS query is performed, otherwise no DNS query is performed[1].

The method `toByteArray` returns a representation of the address as a byte array. This byte array representation contains a header of `RAW_OFFSET` bytes (currently 4 bytes) together with the raw address. The value of the header uniquely describes the address type and allows addresses to be represented as both Java objects and byte arrays. Although all addresses crossing the PPI boundary are `AddressNet` objects, AAs are free use the byte array representation to reduce memory consumption when managing many addresses. The method `fromByteArray` deserializes a byte array representation back into an AddressNet object.

The `getByName` method returns an address for a host by performing a DNS lookup on the given name `host` using a particular address family `family`. If `host` is `null` then the address of the local host is returned.

[1] This method is useful for disabling DNS lookup and consequently prevents threads from blocking each other. Currently, our native code does not use a reentrant version of the C function gethostname and therefore when two threads call getHostName, the second thread must wait until the first thread completes.

5.5.1 IPv4 and IPv6 Addresses

IPv4 and IPv6 addresses are represented as subclasses of the base class `AddressIP`. IP addresses are constructed with the one of the following methods:

```
AddressIPv4(byte[] data, int offset)
AddressIPv6(byte[] data, int offset)
AddressIPv4(byte[] addr)
AddressIPv6(byte[] addr)
AddressIPv4(InetAddress addr)
```

The first constructor constructs an IP address given the *raw* address as a byte array. The first byte of the raw address is expected at position `offset` in the `data` buffer. The second constructor constructs an address object from the byte array representation (which includes the header mentioned above).

The method `toInetAddress` returns the JDK's representation for the given address.

There are two methods that map a given network address to a hostname. The method `getHostName` was described in the base class `AddressNet`. The second method, `toString`, returns a dotted-decimal notation of the given address; no DNS query is performed.

5.5.2 VNET Addresses

The `AddressVnet` class is the base class of all addresses used by the VNET protocol stack. If the AA wants to use VNET then it must construct addresses that are described in this section. Note that when doing network I/O one cannot use VNET address and a IP address in the same I/O operation (eg. having a VNET source address and a IP destination address will return an error).

`AddressVNet` is the base class of three abstract address classes, each of which represents a network layer: `AddressL2`, `AddressL3` and `AddressL4`. The following concrete types are defined: `AddressVN`, which represents a network layer address and `AddressVT`, which represents a transport layer address. The constructors of these two address types are:

```
AddressVN(int addr)
AddressVT(int addr)
```

6 User Application API

A local user application (UA), which may or may not be active, can communicate and/or invoke an active network service implemented as an ASP AA. The interprocess communication (IPC) between a UA and ASP currently uses TCP, a reliable full-duplex byte stream.

For simplicity, this IPC mechanism is mapped into the PPI using the network I/O abstraction using input channels defined by the "api" protocol specification. Thus, API messages are received by an AA using the normal `receivePacket` upcall, where the `chan` parameter is an `InChannel` object and the message data are bytes received from the UA. The api input channel extends the basic channel class to provide full-duplex functionality. The AA returns a reply to the UA by using the `sendPacket` method of this full-duplex channel, casting it into an OutChannel.

API channels cannot be constructed explicitly by an AA; they are constructed implicitly by the ASP EE when a UA contacts the AA through the UA API. A single AA may receive requests from multiple UAs; each creates an implicit API InChannel.

7 Inter AA communication channel

An active application (AA) can communicate and/or invoke an active network service implemented as an ASP AA.

For simplicity, this IPC mechanism is mapped into the PPI using the network I/O abstraction using output channels defined by the "ipc" protocol specification. Thus, messages are received by an AA using the normal `receivePacket` upcall, where the `chan` parameter is an `InChannel` object and the message data are bytes received from the writer AA. The inter AA channel extends the basic channel class to provide a unidirectional pipe from the writer AA to reader AA. The reader AA in turn may open a pipe to the calling writer AA, if it wants to share information. Thus, two channels are required, one in each direction, for creating a full duplex connection between two AAs.

Any number of inter AA channels may be created by an AA, and multiple pipes can also be created between two AAs by specifying unique string names for the channel in the AddressSpec field. String names serve the same purpose as port numbers for other protocols but are more intuitive when used to specify a pipe that is used for communication between AAs. In addition, there is no restriction on the creation of a pipe in which the writer and reader are the same AA.

The AddressSpec field will be used to name instances of a pipe. The AAspec should be mentioned directly in the address spec of a channel open call. Currently, the DemuxiplexKey will not be supported with the pipe.

It is assumed that the names (AAspecs) of cooperating AAs have been communicated to one or both parties out of band. Future work may provide higher level mapping functionality to map services names to the AAspecs of AAs that provide a given service.

When a channel is closed by the writer AA, the reader AA will receive a receivePacket() upcall with length equal to -1. This is the equivalent signal already generated for closed TCP connections in the channel stack. If the reader AA performs a close, the writer will receive an error upon the next write. An asynchronous notification to the writer AA would require a new signal mechanism in the EE.

Pipe data is always passed by value, so the AA's are isolated from each other. An AA also can reject a receivePacket upcall if it does not wish to share any information.

8 Interface to Routing

The PPI class provides an integrated interface to routing information for both the VNet and INet protocol stacks. The PPI class supports the *Routing Support for Resource Reservations* (RSRR) interface, which is a standard interface to routing developed in conjunction with RSVP [8].

8.1 Add a Route

The method AddRoute provides facility to the AA to install an VNet route in the routing table maintained by the EE. Currently there is no support for installing an INet route and an exception is thrown for the same.

```
void    AddRoute(RouteNet route)
```

- route parameter contains the routing information to be added. Details about the RouteNet object are provided below.

```
AddressNet getDstAddress()
void       setDstAddress(AddressVN x)
AddressVN  getInfAddress()
void       setInterfaceAddress(AddressVN x)
AddressVN  getNextHopAddress()
void       setNextHopAddress(AddressVN x)
```

The get methods are used to query the entries while the set methods are used to change the corresponding values. The getDstAddress method returns the destination address for this fowarding rule, while the setDstAddress method sets the new destination address.

The methods explained ahead are specific to RouteVN, which contain details about VNet routes. The getInfAddress method returns the interface address that is used for forwarding to this destination. The getNextHopAddress method returns the next hop address for this forwarding rule.

8.2 Delete a Route

The method DelRoute provides facility to the AA to remove an VNet route from the routing table maintained by the EE. Currently there is no support for removing an INet route and an exception is thrown for the same.

```
void    DelRoute(RouteNet route)
```

- `route` parameter is the route that an AA wants to remove from the routing table.

8.3 Route Query

The method `QueryRoute` provides faciltity to the AA to query an route for a particular AddressVNet destination. Currently there is no support for quering a route to an INet destination and an exception is thrown for the same.

```
RouteNet    QueryRoute(AddressNet addr)
```

- `addr` parameter is the destination address for which a route is required.
- Return RouteNet object containing details about the route requested

8.4 Route Change Notification

The method `RouteChangeNotification` provides faciltity to the AA to be informed whenever there is any route change. It currently only supports notifications for VNet route changes. This is a blocking call and the control is returned to the AA whenever there is any change in the EE's VNet route table. The AA's should use a separate thread if they would like to receive route change notifications and avoid blocking.

```
void    RouteChangeNotification()
```

8.5 Outgoing Interfaces Query

The method `QueryOutgoingInterfaces` requests the current list of outgoing interfaces for a given destination address `dst` and perhaps a source address `src`. It supports queries for VNet and INet multicast addresses only.

The method provides synchronous replies for VNet addresses, but will provide deffered synchronous replies for a INet muticast address request. If an AA would like to avoid this deferred synchronous response, it should use a separate thread for multicast address queries.

As there is currently no support for outgoing interface change notifications for VNet destination address, the `keepEntry[0]` must be set to false, else an exception is thrown for the same. Please note that the queries for INet unicast destination addresses are currently not supported.

```
Vector      QueryOutgoingInterfaces(AddressNet src, AddressNet dest,
                    int[] in_if, boolean[] keepEntry, Object reference)
```

- `src` parameter is a source address for which a list of outgoing interfaces is required. If the host's routing protocol does not require a source addresses then set `src = null`.
- `dst` parameter is the destination address (either VNet or INet multicast) for which a list of outgoing interfaces is required.
- `in_if` parameter returns the incoming interface, if appropriate for the routing protocol in use. The value of `in_if[0]` is ignored by the EE and it is only used as a return parameter.
- `keepEntry` parameter ensures that state is kept in an internal cache so that further asynchronous route changes can trigger an notification to an interested AA. If `keepEntry[0]` is `false` then the relevant entry is removed from the cache and consequently any further changes in the given route will not result in any notifications. On return, `keepEntry[0]` is `true` if and only if the routing process can provide the outgoing interfaces list change information. If `keepEntry[0]` is false then the AA must re-call this function to elicit another change notification. Otherwise, if `keepEntry[0]` is true then the AA can call the new `OutgoingInfChgNotification` downcall mechanism for receiving further notifications.
- `reference` parameter may be used to associate a particular outgoing interface list change notification with the corresponding request. Internally this parameter is treated as an opaque object.
- Return `Vector` of `Integer`s containing the outgoing interface LINs

8.6 Outgoing Interface Change Notification

The method `OutgoingInfChgNotification` provides notification to AA whenever there is any change in the outgoing interface list for an source-destination pair. It supports notification for only INet multicast destination address. Currently there is no support for notifications in case of changes in the VNet interface list for a source-destination pair.

This is a blocking call and the control is returned to the AA whenever there is any change in the outgoing interface list for which the AA had requested notification. The AA's should use a separate thread if they would like to receive interface change notifications for a source-destination pair and avoid blocking.

```
Vector    OutgoingInfChgNotification()
```

- Return Vector of `RouteChgObj` objects which contains the details about the change in the outgoing interfaces. One can query the methods listed below of the `RouteChgObj` to obtain details about the change.

```
AddressNet getSrc()
AddressNet getDst()
int getIif()
```

```
Vector getOif()
boolean isNotifiable()
Object getObject()
```

The `getSrc` method returns the source address for which the outgoing interface has changed. The `getDst` method returns the destination address for which the outgoing interface has changed. The `getIif` method returns the incoming interface LIN for this source-destination pair. The `getOif` method returns a `Vector` of `Integer` which specify the outgoing interface LINs for this source-destination pair. The `isNotifiable` method returns `true` if further notification can be provided. The `getObject` method returns the parameter which may have been specified in the `QueryOutgoingInterfaces` method to associate a particular outgoing interface list change notification with the corresponding request.

9 Interface to Traffic Control

This `TrafficControl` class provides an interface to the kernel's traffic control module, which classifies packets into flows and enforces admission control. The `TrafficControl` class provides a Java wrapper around native methods, which call kernel traffic control functions. Use of these methods assumes that the EE has been configured with native methods turned on. For more information about the kernel traffic control interface see the section entitled "RSVP/Traffic Control Interface" in [4]. In the text below, we use the terms *session*, *flowspec* and *adspec*, which are all described in [4].

The `TrafficControl` class exports the following methods:

```
static int     init(InterfaceNet inf)

static int     addFlowspec(InterfaceNet inf, byte[] flowspec,
                       byte[] Path_Te, byte[] adspec,
                       int flags)

static int     delFlowspec(InterfaceNet inf, int rhandle)

static int     modFlowspec(InterfaceNet inf, int rhandle,
                       byte[] flowspec, byte[] Path_Te,
                       byte[] adspec, int flags)

static int     addFilter(InterfaceNet inf, int rhandle,
                       byte[] sess, byte[] filt)

static int     delFilter(InterfaceNet inf, int fhandle)

static byte[]  advertise(InterfaceNet inf, byte[] adspec,
                       int flags)
```

9.1 Initializing an Interface

The `init` method allows the kernel traffic control module to initialize state for the given interface. The method returns `TC_ERROR` if the traffic control module will be unable to regulate traffic over this interface, otherwise `TC_OK`.

9.2 Creating a Reservation

A kernel reservation is made with the `addFlowspec` method. This method calls the kernel to make a reservation for a flow. It first checks admission control, and if the admission test succeeds, makes a

reservation. If the reservation is successful, the method returns a handle, called the `rhandle`, for the reservation, otherwise it returns `TC_ERROR`. The `rhandle` is used by the caller for future references to this reservation. The following parameters are used:

- `inf` is the interface to be used,
- `flowspec` is the Flowspec as a byte array,
- `Path_Te` is the SenderTspec as a byte array,
- `adspec` is the Adspec as a byte array,
- `flags` contain the following police flags: `TCF_E_POLICE`, `TCF_M_POLICE` and `TCF_B_POLICE`.

9.3 Deleting a Reservation

The `delFlowspec` method deletes an existing kernel reservation. This routine deletes the flow for the specified handle `rhandle`, which was given to the caller by the `addFlowspec` method. The `inf` parameter is the interface to be used and the `rhandle` parameter is the reservation handle for the traffic flow. The method also deletes all corresponding filter-specs. If successful, the method returns `TC_OK`, otherwise it returns `TC_ERROR`.

9.4 Modifying a Reservation

The `modFlowspec` method modifies an existing kernel reservation. The parameters for this method are identical to the `addFlowspec` method except for the addition of `rhandle`, which was returned by the `addFlowspec` method. The method returns `TC_OK` if successful otherwise `TC_ERROR`.

9.5 Adding a Filter

Once a reservation has been made with `addFlowspec`, the method `addFilter` is used to associate an additional filter-spec with the reservation specified by `rhandle`. If successful, the method returns a handle, called `fhandle`, otherwise it returns `TC_ERROR`. The parameters are:

- `inf` is the interface to be used,
- `rhandle` is the reservation handle for the traffic flow,
- `sess` is the session given by the RSVP-packet SESSION object (including object header) in network byte order,
- `filt` is the Filterspec given by the RSVP-packet FILTERSPEC object (including object header) in network byte order,

For more information on RSVP-packet object formats, see Appendix A in [4].

9.6 Deleting a Filter

The `delFilter` method removes an existing filter associated with a reservation. The `inf` parameter is the interface to be used and the `fhandle` parameter is the handle given by `addFilter`. If successful, the method returns `TC_OK` otherwise it returns `TC_ERROR`.

9.7 Updating the Adspec

The *One Pass With Advertising (OPWA) Adspec* is an object sent with a RSVP path message to collect information that receivers can use to predict the end-to-end service [7, 4]. The adspec object is typically opaque to applications, and the kernel provides the `advertise` method to modify it. If successful, the method returns the modified adspec as a byte array otherwise it returns `null`. The parameters are:

- `inf` is the interface to be used,
- `adspec` is the Adspec object as an opaque byte array,
- `flags` includes the flag `PSBF_NonRSVP`, which is set if there is at least one non-RSVP node between the previous RSVP hop and this node.

10 Logging and Debugging

10.1 The asp.lib.LogWriter class

The `asp.lib.LogWriter` manages log files and enforces an upper bound on the size of each opened log file. Once an instance of `LogWriter` is created, the `write` methods are used to write data to the log file. When the upper bound is reached, `LogWriter` closes the file, renames the file (overwriting any previously renamed file) and then opens a new file for writing.

We call the open file the *primary file* and the renamed file the *secondary file*. By always overwriting the secondary file, `LogWriter` ensures that any instance consumes finite disk resources. The `LogWriter` constructor allows the user to determine the total disk space (in bytes) that may be consumed by the instance.

The `LogWriter` class extends the `java.io.Writer` class, which has the advantage that the *existing* classes (in the `java.io.Writer` hierarchy) can be used to add functionality to `LogWriter`. An

example class is provided in appendix 11 which provides additional functionality such as time-stamping and finer-grained control of debugging output.

The `LogWriter` class provides one addition method (actually a constructor) to the abstract `Writer` class:

```
LogWriter(String primary_filename,
        String secondary_filename, int max_size)
```

This constructor opens a primary log file `primary_filename` and enforces an upper bound on its size given by `max_size`. When *half* this upper bound is reached, the primary file is renamed to the secondary filename. Note that the total size of both log files will consume no more than the upper bound as specified by the `max_size` parameter.

11 The asp.lib.Debug class

The `asp.lib.Debug` class provides functionality for controlling the granularity of debugging output. The class maintains an internal state called the *debugging level* for a given Java class and method name prefix. The *debugging level* can have one of the values from the following ordered list: `ENR` (non-recoverable error), `ER` (recoverable error), `WLOW` (low volume of warnings), `WHIGH` (high volume of warnings), `DLOW` (low volume of debugging), `DMED` (medium volume of debugging), and `DHIGH` (high volume of debugging). The level `ENR` is the class of non-recoverable errors, the most critical errors, and has the lowest value. The level `DHIGH`, corresponding to high volume debugging messages, has the highest value. When logging a message with one of the `log` methods (see below), a message is printed only if `message level <= debugging level` and the calling Java class and method name matches the associated prefix name string.

The class has two constructors:

```
Debug(java.io.OutputStream out, boolean autoFlush)
Debug(java.io.Writer out, boolean autoFlush)
```

The parameters are:

- `out` is an underlying output stream that `Debug` uses to do the actual writing (examples are `System.out` and any class in the `java.io.Writer` hierarchy).
- if `autoFlush` is true then the output stream is flushed before the `log` methods complete,.

For example, a `Debug` instance can use `stdout` as the underlying output stream:

```
Debug debug = new Debug(System.out, true);
```

or a `LogWriter` instance as the output stream:

```
Debug debug = new Debug(
        new LogWriter("asp.log", "asp.log.bak", 1024*1024),
        true);
```

The `Debug` class provides the following methods:

```
void log(int level, String str)

void log(int level, Object o1)
void log(int level, Object o1, Object o2)
void log(int level, Object o1, Object o2, ...)

void log(int level, byte[] bp, int offset, int len)
void log(int level, Object o1, byte[] bp, int offset, int len)
void log(int level, Object o1, ..., byte[] bp, int offset, int len)

void verbose(boolean v)

void setFilter(String name, int level)

static int getDebugLevel(char flag, String str)
```

The `setFilter` defines `debugging level` and caller prefix name filters that trigger the printing of debugging messages. The caller of the `log` method is determined by stack inspection and the Java class and method are concatenated using a period character as separator. For example, a valid caller name string is "asp.lib.Debug.setFilter". The prefix string supplied to `setFilter` is matched against the caller name to determine if it the beginning sequence of characters in the caller name string matches. Matches are then logged if the `message level <= debugging level`. Here are some examples:

- "asp.lib.Debug.setFilter" - Match only messages generated by the `setFilter` method of the Java class `asp.lib.Debug`.
- "asp.lib.Debug" - Match all messages generated by the the Java class `asp.lib.Debug`.
- "asp.lib" - Match all messages generated by the the Java classes in the package `asp.lib`.
- "as" - Match all messages generated by the the Java classes in all package that start with `as`.
- "" - Match all messages generated by all classes and all methods.

The `log` methods provide the main functionality for this class. The `log` methods conditionally logs a message based on the the message `level` parameter and the calling Java class and method name. In particular, a debugging message is displayed only if `message level <= debugging level` and the calling Java class and method name match the prefix that was specified by a previous call to the

`setFilter` method. By design, the number of debugging messages can be reduced by increasing the message level. The third set of `log` methods is useful for displaying the contents of data packets in hexadecimal format.

In general, it is more efficient to pass multiple object arguments to the `log` methods rather than producing a string concatenation of the message prior to the call. This is because the debugging message may or may not get logged depending on the filters. Producing a string concatentation of the message prior to the call generates useless work when the message is suppressed. But there is still a cost for marshalling the arguments.

The `verbose` method is used to set the verbose mode on and off. If the verbose mode is on, debugging messages will indicate the calling class and method name.

Finally, the `getDebugLevel` method is used to translate command line arguments to debugging level values.

12 Acknowledgments

The ASP EE was implemented by the ARP-Team: Bob Braden, Alberto Cerpa, Jeff Kann, Bob Lindell, Ted Faber, Graham Phillips, Vivek Shenoy and Ya Xu. Comments or questions may be sent to `mailto:isi-arp@isi.edu`.

A Configuring the ASP EE

This section describes the three configuration files that may be necessary to execute the ASP EE.

A.1 asp.conf File

The file `files/asp/asp.conf` specifies general configuration information for the ASP EE. It specifies the input channels that are opened by the EE, the AAs that are to be loaded when the EE starts ("boot time"), which AAs should be loaded at boot time, and other control information.

The configuration file consists of a series of entries with the following syntax.

```
<entry>          ::= <channel> <newline> | <bootstrap> <newline> |
                     <debug> <newline> | <include> <newline> |
                     <datagramSocket> <newline> | <netbuffersize> <newline> |
                     <netiod> <newline> | <fileserverport> <newline> |
                     <separateclassloaders> <newline> | <rdpport> <newline> |
                     <defaultdemuxport> <newline> | <comment> <newline>

<channel>           ::= InChannel <space> <filter spec> space
                                      invokes <space> <AA entry>

<include>           ::= Include <space> " <filename> "
<datagramSocket>    ::= DatagramSocketQueue <space> [ on | off ]
<rdpport>           ::= RDPPort <space> <integer>
<netbuffersize>     ::= NetBufferSize <space> <integer>
<fileserverport>    ::= FileServerPort <space> <integer>
<defaultdemuxport>  ::= DefaultDemuxPort <space> <integer>
<separateclassloaders>  ::= SeparateClassLoaders <space> [ on | off ]

<filter spec>       ::= <protocolSpec> <space> <addressSpec> { <space>
                                                  <DemultiplexKey> }

<protocolSpec>      ::= api/asp | <VNET protocolSpec> | <INET protocolSpec>

<VNET protocolSpec> ::= vif<integer>* | vif<integer>*/vn/vt | vif<integer>*/vn/vt/asp |
                        vif<integer>*/vn/rdp | vif<integer>*/vn/rdp/asp

<INET protocolSpec> ::= if/ipv4 | if/ipv4/asp | if/ipv4h | if/ipv4h/asp |
                        if/ipv6 | if/ipv6/asp | if/ipv4/udp | if/ipv4/udp/asp |
                        if/ipv6/udp | if/ipv6/udp/asp |
                        if/ipv4/tcp | if/ipv4/tcp/asp

<addressSpec>       ::= <vt addressSpec> | <UDP addressSpec> | <IP addressSpec>
```

```
<vt addressSpec>    ::= <ToS> : <remoteAddr> [ : <localPort>
                                    [ : <localAddr> [ : <remotePort> ] ] ]

<UDP addressSpec>   ::= <remoteAddr> [ : <localPort>
                                    [ : <localAddr> [ : <remotePort> ] ] ]

<IP addressSpec>    ::= <protocol> [ : <remoteAddr> [ : <localAddr> ] ]

<ToS>               ::= TOS | E2E
<localAddr>         ::= <integer> | <IPv4 addr> | [<IPv6 addr>] | *
<localPort>         ::= <integer> | *
<remoteAddr>        ::= <integer> | <IPv4 addr> | [<IPv6 addr>] | *
<remotePort>        ::= <integer> | *

<DemultiplexKey>    ::= [<operator> <space>] <offset> <space> <length> <space>
                                                        <mask> <space> <value>
<operator>          ::= ! | = | < | <= | > | >=
<offset>            ::= <unsigned 16-bit number>
<length>            ::= <unsigned 16-bit number>
<mask>              ::= 0x <hex-digit> [ <hex-digit> ]
<value>             ::= 0x <hex-digit> [ <hex-digit> ]

<AA entry>          ::= * | AAname <space> <Q-AAname> | Legacy <space> <Q-AAname>
<Q-AAname>          ::= " <AAname> "

<bootstrap>         ::= Bootstrap <space> <AAname>

<netiod>            ::= Netiod <space> [on|off]

<debug>             ::= Debug <space> <output>
<output>            ::= file [on|off] | file on|off " <filename> " |
stdout [on|off]

<comment>           ::= # { <any character except newline> }

<AAname>            ::= <AAnameChar> | <AAname> <AAnameChar>
<AAnameChar>        ::= <letter> | <digit> | ! | # | $ | % |
                         & | = | + | - | _ | @
<filename>          ::= <filenamechar> | <filename> <filenamechar>
<filenamechar>      ::= <AANameChar> | { | } | $
```

Here `<AAname>` is case sensitive, but all other fields are case-insensitive.

The following types of entries are defined.

InChannel

Specifies an EE input channel to be created when the EE starts.

- `<protocolSpec>`
  Specifies a protocol stack for processing packets in the input channel. ProtocolSpecs that include the `asp` protocol, eg. `vif/vn/vt/asp`, must have a non-wild AAName.
- `<addressSpec>`
  Binds input packet header fields specified in the `<protocolSpec>`. Two protocolSpecs that differ only by the `asp` protocol may not share the same destination port. For example, one cannot open `vif/vn/vt E2E:*:9999 RSVP` and `vif/vn/vt/asp E2E:*:9999 RSVP` concurrently. However the same destination port may be shared by protocolSpecs that include the `asp` protocol. For example, one can open `vif/vn/vt/asp E2E:*:9999 RIP` and `vif/vn/vt/asp E2E:*:9999 RSVP` concurrently.
  The `<addressSpec>` also determines the default addressSpecs for the specified AA (assuming that the `<AA Name>` is not wild (i.e., asterisk). If a particular AA has more than a one channel entry then the *first* matching entry is used to set the default addressSpec values (for that AA).
  The following default `<addressSpec>` field values are used when the first AAname match is with a wildcard AAname (i.e., asterisk):
  1. `<ToS> = HBH`,
  2. `<localAddr> = ANY_ADDRESS`,
  3. `<remotePort> = ANY_PORT`,
  4. `<remoteAddr> = ANY_ADDRESS`,
  5. `<localPort> = ANY_PORT`.
- `<demultiplexKey>`
  Specifies a packet filter that the packet must match, after the protocol headers listed in `<protocolSpec>` have been processed and removed.

The EE must determine the AAspec before the packet can be delivered to the corresponding AA. The `invokes <AA entry>` clause defines one of three ways to determine an AAspec for the given EE InChannel.

1. Wildcard (ie. `*`): the *AASpec* is extracted from the EE packet header. Because the *AASpec* is extracted from the EE packet header, the specified channel must include `asp` in the protocolSpec, for example, `vif/vn/vt/asp`. Thus, only those channels that include `asp` in the protocolSpec can support this wildcard case.
2. `AAname` keyword: all packets arriving on the channel are directed to the specified AA. If the protocolSpec does not include `asp` then there may be at most one of these channels associated with a given destination port. Multiple channels may share a given destination port as long as all these channel include `asp` in their protocolSpecs.
3. `Legacy` keyword: the AA associated with the `AAname` is loaded and its `extractAASpec` method is called. The returned *AASpec* then determines the AA to which the packet is delivered.

Bootstrap

A Bootstrap entry implies that the particular AA is to be loaded when the EE starts. The EE loads the AA named `<AAname>`, obtaining the rest of the *AAspec* (the *AAbase* name and the path) from the `AAspec.conf` file (below).

Netiod

A `Netiod` entry specifies whether the EE should use Netiod for opening RawIP and UDPv6 InChannels and OutChannels. If this option is turned off, the above mentioned channels cannot be opened by the ASP EE.

FileServerPort

A `FileServerPort` entry specifies the TCP/RDP port on which the EE listens to requests for classes.

DatagramSocketQueue

A `DatagramSocketQueue` entry enables internal EE queueing used in congestion control experiments.

SeparateClassLoaders

By default, all AAs share a single Java class loader. This conserves the memory footprint of AAs by allowing shared code to be loaded once for multiple AAs. This is discussed in more detail in the section on class loading in [3]. Turning `SeparateClassLoaders` on puts each AA in a separate Java class loader. This is useful in environments where AAs will be created and destroyed often and it is important to obtain garbage collection of class definitions. Most Java implementations currently garbage collect class definitions only when the containing class loader is also garbage collected. When AAs are terminated, and this option is on, the AAs class loader will be unreferenced by the EE and become a candidate for garbage collection along with the class definitions it contains.

Debug

A `Debug` entry specifies whether the EE should write any logging output. By default the EE writes to stdout and to a primary log file, called `asp.log`. Once the log file reaches a certain size, it is swapped out to `asp.log.bak`, and a new primary file is opened. Output to stdout and to the log files may be turned off using the `Debug` keyword.

If a filename is given after the debug file on—off sequence, a log file with that name is used instead of `asp.log`, if possible. `${HOSTNAME}` is expanded in teh filename as it is for the include directive.

Include

An `Include` entry includes another configuration file verbatim at this point in the configuration processing. The filename should be in double quotes and must reside in the EE file space. If the string `${HOSTNAME}` appears in the filename, it is replaced with the EE hostname (see below).

NetBufferSize

A `NetBufferSize` entry configures the default NetBuffer size used by the EE to be the integer given. The default netbuffer size is 64KB, large enough to accept the biggest UDP packet. This conservative size is chosen because ASP will silently drop packets smaller than the largest configured NetBuffer size. While it is often safe to reduce this number, and can improve performance to do so, changing the value introduces the potential for confusing behavior. Change this value only if you understand the ramifications.

RDPPort

An `RDPPort` entry configures the UDP port used to tunnel RDP messages between ASP instances. Like the `FileServerPort`, all ASP nodes wishing to communicate via UDP-tunneled RDP must use the same RDP port. This entry must be present to use UDP-tunneled RDP.

DefaultDemuxPort

A `DefaultDemuxPort` entry specifies the INET/VNET port on which the EE has opened an wildcarded InChannel (the *AASpec* is extracted from the EE packet header). When an AA wants to send a packet to another instance of an AA running on a different ASP node, it needs to know the destination port on which the ASP EE has opened a wildcarded InChannel. This entry is used to specify that default demux port value for the ASP EE. Thus all the ASP nodes in the same topology and wishing to communicate with each other must use the same default ASP demux port.

An example asp.conf configuration file is as follows:

```
##
## Open three channels for demultiplexing on the AASpec in
## the ASP EE packet header.
##
InChannel api/asp E2E:*:8989 invokes *
InChannel vif/vn/vt E2E:*:9999 invokes *
InChannel vif/vn/vt HBH:*:9999 invokes *
##
## The default TCP port for serving Java class files is 2777.
## Uncomment the line below and change this number if required.
##
## FileServerPort 2777
##
## The default demux port for the ASP nodes in a topology is 9999.
## Uncomment the line below and change this number if required.  It should be
## set to the port number on which the wildcarded InChannels are open on all
## ASP nodes in the same topology.
##
## DefaultDemuxPort 9999
##
## Don't log to Stdout
##
Debug Stdout Off
```

```
##
## Allow RDP to be tunneled on UDP port 15000
##
RDPPort 15000

##
##
## Separate Class Loaders for each AA
##
SeparateClassLoaders On

##
## Load local configuration
##
include "asp.${HOSTNAME}.conf"
```

A.2 AAspec.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 an AAspec is described in an Appendix of [3].

A.3 hostname file

The `hostname` file sets a string value that this EE will use as a unique identifier. This document sometimes refers to this identifier as the ASP hostname. The ASP hostname is used to expand `${HOSTNAME}` in `asp.conf` and scoping in the VNET configurations. The `hostname` consists of a single line containing the identifier. Other lines are ignored.

If no `hostname` file is present, the Domain Name System is consulted for the name of the local host, and that name is used. This should be avoided because of the inherent ambiguities arising from hosts with more than one DNS name or IP interface. If neither the hostname file nor a DNS name is available, the string “noone” is used.

A.4 VNET Configuration

This section describes the two configuration files required for virtual network connectivity, provided by the VNET package in the ASP EE. These two files define the initial virtual network interfaces and the initial route entries when the EE is booted. These route entries are used for default forwarding in VNET's internetwork layer. After boot time, AA operation may modify this information dynamically, adding and deleting virtual interfaces and/or updating routing entries.

It is important to note that there must be a consistent set of configuration files deployed on the nodes of the virtual topology. Inconsistent address and port number assignments to virtual interfaces may silently black hole packets.

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 shared by the machines `bro.isi.edu` and `son.isi.edu`. `Bro.isi.edu`'s ASP hostname is `bro` and `son.isi.edu`'s ASP hostname is `son` (see the above section on the hostname file).

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

Each entry specifies a point-to-point link in the virtual topology, implemented using UDP encapsulation. For each virtual link, local and remote end-points are specified. For each end point, there is a VNET internetwork layer address (integer) followed by a corresponding `IP address/UDP port` item. The end-point of the first link is `any/1111` and `obelix.cs.sun.ac.za/2222`. The keyword `any` means any IP address of the local machine.

In order to have the ASP EE run on top of Anetd [6], one must specify a VNET virtual link layer with ANEP demultiplexing and UDP/IP encapsulation. Such packets are framed using both a UDP/IP header and an ANEP header, but no VNET-specific header. Virtual link layer addresses for this layer use AddressAnep objects, which contain an IP address, UDP port number, and the ANEP Type ID [1]. For each end point, there is a VNET internetwork layer address (integer) followed by a corresponding `IP address/UDP port/ANEP Type ID` item. Packets arriving from Anetd appear on the standard input to the JVM rather than a particular UDP port number. By specifying port 0 for the local end-point a LayerAnep (see Section 4.3.7 in [3]) object instance can be created to listen on standard input. All other non-zero port numbers cause LayerAnep to listen on the corresponding UDP port number.

The VNET internetwork layer addresses must be unique for each (IP address, UDP port) pair. Note that there may be multiple VNET addresses associated with a single remote IP address, as is the case with machine `son`. In addition, UDP ports may safely be shared by more than one VNET interface.

The scoping lines, e.g., `son:`, restrict the following entries to apply only to the machine with that hostname. `Bro.isi.edu` will use only the first 5 lines (below the `son:` line) because its ASP hostname is `bro`. Similarly `son.isi.edu` will only use the last 2. Lines before the first scoping line will be read by any machine reading the file. Although the same rules apply to routing tables, there are no such lines in the example.

The following route table example contains 3 route entries:

```
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 entries individual routes to each remote VNET address from the local VNET address. When running the routing protocol RIP, explicit route table entries are only needed for specifying additional static routes. RIP runs by default in the ASP EE distribution so that users can use empty route table configuration files in the usual cases.

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

```
<interfaceTable>    ::= { <infEntry> <newline> | <commentLine> <newline> |
                          <scope> <newline> }

<infEntry>          ::= <infClass> <space> <addrClass> <space> <localVNAddr>
                        <space> { <localUDPAddr> | <localUDPAddrANEP>  }
                        <space> <remoteVNAddr>
                        <space> { <remoteUDPAddr> | <remoteUDPAddrANEP> }

<infClass>          ::= asp.InterfaceVN

<addrClass>         ::= asp.AddressUI | asp.AddressAnep

<localVNAddr>       ::= <VNAddr>
<remoteVNAddr>      ::= <VNAddr>
<VNAddr>            ::= <32-bit integer>

<localUDPAddr>      ::= <IPAddr> / <UDPport> | any / <UDPport>
<localUDPAddrANEP>  ::= <IPAddr> / <UDPport> / <ANEP_TypeID> |
                         any / <UDPport> / <ANEP_TypeID>
<remoteUDPAddr>     ::= <IPAddr> / <UDPport>
<remoteUDPAddrANEP> ::= <IPAddr> / <UDPport> / <ANEP_TypeID>
```

```
<IPAddr>            ::= <hostname or numeric IPv4/IPv6 address>
<UDPport>           ::= <unsigned 16-bit number>
<ANEP_TypeID>       ::= <unsigned 16-bit number>
<scope>             ::= { any letter, number, or .} :

<commentLine>       ::= # { any character except newline }

<routeTable>        ::= { <routeEntry> <newline> | <commentLine> <newline> |
                          <scope> <newline> }

<routeEntry>        ::= <routeClass> <space> <localVNAddr> <space> <remoteVNAddr>

%% <routeClass>     ::= <Java class implementing +asp.RouteL3>
<routeClass>        ::= asp.routeVN
<scope>             ::= { any letter, number, or .} :
```

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] S. Berson, B. Braden, and L. Ricciulli. Introduction to the abone. `http://www.isi.edu/abone/DOCUMENTS/ABarch/`, 2000.

[3] Bob Braden, Alberto Cerpa, Ted Faber, Bob Lindell, Graham Phillips, and Jeff Kann. *Introduction to the ASP Execution Environment*, 2000. `http://www.isi.edu/active-signal/ARP`.

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

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

[6] Livio Ricciulli. Anetd: Active networks daemon (v1.0). `http://www.csl.sri.com/ancors/anetd/docs/`, 1999.

[7] S. Shenker and L. Brelau. Two aspecs of reservation establishment. In *ACM SIGCOMM '95*, 1995.

[8] D. Zappala and J. Kann. A routing interface for RSVP. `http://www.isi.edu/~kann/Publication/draft-ietf-rsvp-routing-02.txt`, November 1998.