A Programmer’s Tutorial for the ASP Execution Environment (Release 1.6)

Graham Phillips, Bob Lindell
USC/Information Science Institute
graham@isi.edu, lindell@isi.edu

February 18, 2003

Contents

1 Introduction 1

2 Working with Network Addresses 2

3 Launching an Active Application (AA) 3

4 The Anatomy of an Active Application (AA) 4
   4.1 The AContext 4
   4.2 Activities 4
   4.3 Receiving Packets 4
   4.4 Sending a response to the UA 6
   4.5 Sending a packet to another instance of this AA 6

5 Putting it all together: a complete Ping AA 7
   5.1 Ping AA 8

A Availability of ASP EE source code 11

1 Introduction

This tutorial provides examples for programmers who wish to develop Active Applications (AAs) using the ASP Execution Environment [1]. The ASP Execution Environment (ASP EE) is an active-network environment for executing network protocol code written in Java. AAs execute under the ASP EE similar to the way applications execute under a conventional operating system. In place of a system call interface, the ASP EE provides active protocol code with a Protocol Programming Interface (PPI).
AddressNet addr = AddrUtil.String2AddressNet("asp-private-vnet://1")
if (addr == null) {
    // Something wrong with the encoded address.
}

Figure 1: Decoding an address from a string to a Java object

The PPI provides a NodeOS like channel interface [2] to the two network protocol stacks in ASP EE: a virtual network stack called VNet and the IP stack. The VNet stack supports the construction of a virtual network over UDP/IP point-to-point links and the transmission of VNet packets. The IP stack requires native method support (i.e. C code) when using anything other than TCP or UDP. The examples in this tutorial use the VNet stack.

The example presented in the document is the network Ping application. This application is very common on hosts that provide an IP networking implementation. From a given host in the network, Ping can be used to echo a reply from another host in the network. This reply is used to determine whether the other host is reachable and to compute the approximate round trip delay between the two hosts.

2 Working with Network Addresses

In this section we describe how to use network addresses in the ASP EE. All address objects that pass across the AA/EE boundary extend the AddressNet class. Currently this class is divided into two main sub-classes, a class AddressIP that implements IPv4 and IPv6 addresses, and a class AddressVNet that implements VNet addresses. In all network I/O operations (ie. send and receive), it is important that compatible address types are used for all address parameters. Supplying incompatible address types to PPI functions may generate an error.

Network addresses will typically be passed using URL style strings between the User Application (UA) and AA. Figure 1 shows an example code fragment that decodes a URL style address string into an AddressNet object.

Another common operation on addresses is the encoding and decoding of addresses into byte array format. These operations are useful for protocol implementations that generate packets with embedded address information in a binary form.

A method to decode network addresses from a byte array into an AddressNet object is provided by the AddressNet.fromByteArray method.

Address objects can be encoded using the toByteArray implemented by every address object. The encoding includes a header which describes the type of address, followed by several bytes that contain the raw address. The length of the encoded address is dependent on the address type, which is one of IPv4, IPv6, and VNet.
Typically, an AA will be launched by a request originating from a UA. The ASP EE has defined a protocol for the UA and the AA to communicate with each other. See [1] for further details. Currently, the UA protocol is layered on TCP, allowing a UA to communicate with an AA on any host that has IP accessibility.

We have provided a general purpose UA that might be suitable for most AA developers. The Rsh UA provides functionality similar to a remote shell. It passes the command line arguments of the UA to the AA when the UA is invoked. Subsequently, it passes line oriented text between the UA and the AA. The Rsh UA will terminate when it receives an end of file (EOF) indication from the AA. It can also be terminated by the user using standard command line interruption or termination signals. If this UA functionality is not sufficient for a newly developed AA, the AA developer will need to develop an appropriate UA.

The following example shows how to invoke the Ping AA using the Rsh UA. The arguments to Rsh are AASpec, IP host, port number, and optional AA specific command line arguments. The first three arguments, AASpec, IP host, and port number, instruct the Rsh UA to send a UA request to the ASP EE on the IP host using the given AASpec. Any additional command line arguments are passed to the AA and are not interpreted by the Rsh UA program.

The AASpec stands for an AA Specification and includes the following three pieces of information: (1) the name of the AA, (2) the search path where the AA byte code is located and (3) the name of the class that implements the AAContext. The syntax and the semantics of the AASpec are further described in [1].

Figure 2 shows how a Unix shell command the Ping AA can be invoked on host.isi.edu/8989 to ping the remote host 5 in the VNet topology. The forth argument to Rsh, asp-private-vnet://5, which is an URL style VNet address, is passed to the ping AA and is not processed by Rsh directly. In this manner, any number of additional command line arguments may be provided to the Rsh UA, which will be in turn be passed to the AA.
4 The Anatomy of an Active Application (AA)

4.1 The AAContext

Each active-application (AA) must minimally include a subclass of the AAContext class, which is the base class for all AAs. AAContext is an abstract class and all subclasses must provide an implementation of the receivePacket packet upcall method.

4.2 Activities

It is possible for an AA to concurrently support multiple UA requests. We refer to this concept of multiple concurrent requests as an AA that supports multiple activities. When a UA request arrives at an AA, from somewhere in the network, the AA needs to distinguish this request and packets generated for this request from other requests. One way to distinguish activities is to assign a globally unique activity ID to each request. Subsequent packets sent into the network to other nodes with the same AA will carry this ID and be used to separate out the traffic for a given activity. Because each UA request can be initiated at any node in the network, ID allocation, in general, requires distributed coordination.

The method Activity.create generates a new activity ID. Activity IDs may be sent in a binary form in packets and the method Activity.get regenerates the activity ID from the packet at the receiving AA. Finally, the method destroy is used to deallocate an activity ID when the activity has completed.

4.3 Receiving Packets

When packets are received, the EE upcalls into the AA receivePacket method. Figure 3 shows an example of how an AA might implement the receivePacket method. In this example, the AA distinguishes between those packet that are sent by the UA and those packets that are sent by another instantiation of the AA elsewhere in the network.

The parameters to the receivePacket method are as follows. The first parameter provides a reference to the InChannel that received the packet. The second parameter is the actual data buffer for the packet. The third parameter provides ancillary information about the reception of the packet, some of which is not provided by the NodeOS InChannel specification [2]. This ancillary information includes the incoming network interface, source port, and destination port.

Packets from UAs are distinguished by the InChannel. All UA request packets are received on an InChannel having the “api” protocol specification. Two packets are from the same UA instance if the InChannel references in the receivePacket upcall are identical. The call to the Rsshlib.get method retrieves the existing Rssh context for this packet. All packets arriving from the same UA instance are associated with the same context.
public class Ping extends AAContext {
    public void receivePacket(InChannel c, NetBuffer b, Attrib a) throws IOException {
        String ps = c.getProtocolSpec();
        if (ps.startsWith("api")) {
            Rshlib r = Rshlib.get(c);
            if (b.getLength() == -1) {
                // UA Program quit unexpectedly,
                // cancel existing request
                r.close();
                return;
            }
            String args[] = r.argv(b);
            if (args != null) {
                // Process new UA request with
                // command line arguments
                return;
            }
            // Process additional UA input for
            // an existing request
            return;
        }
        // Process non-UA packet
    }
}

Figure 3: Example code to receive a packet inside an AA
public void receivePacket(InChannel chan, NetBuffer msg, 
        Attrib attr) throws IOException {
    String ps = c.getProtocolSpec();
    if (ps.startsWith("api")) {
        Rshlib r = Rshlib.getC();
        r.sendin("Received UA request");
        r.close();
    }
}

Figure 4: Sending a UA response

When a new UA request arrives, the Rshlib.get method will not find an existing context. In this case, it will create and return a new context. UA contexts should be destroyed when they are no longer needed. In this example, when the AA receives an unexpected termination of the UA, it closes (or destroys) the context after sending an end of file (EOF) indication to the UA. If the UA context is new, it will have command line arguments which were passed from the Rsh UA. These arguments are retrieved using the argv method. Subsequent data packets for an existing UA request will not have command line arguments and the argv method will return null.

Finally, if the packet was not received from an “api” based InChannel, then the arriving packet is a network packet.

4.4 Sending a response to the UA
AAs can send line oriented text responses to the Rsh UA at any time. Text lines are sent using the sendin method. The AA can also send an end of file (EOF) indication to the UA using the close method.

Figure 4 shows an example of how an AA could be written to send back a text response to the invoking UA.

4.5 Sending a packet to another instance of this AA
There are several variations of sending a packet which we will discuss in this section. The AA may choose to source a packet, continue forwarding a packet hop by hop, or send a reply to a packet.

To source a data packet, the AA opens an OutChannel and then calls sendPacket. The AA must provide a data buffer and a destination address as part of the send operation. Figure 5 shows an example of how to source a packet.

In VNet mode, packets may be sent either hop-by-hop or end-to-end. If a packet is sent in end-to-end mode then the packet will arrive at the destination address in the usual way. If, however, the packet is sent in hop-by-hop mode then the packet will be intercepted at every VNet-hop along the way to the final destination. By intercepted we mean that the AA will be instantiated on
AddressNet dest = // A destination address
NetBuffer b = NetBuffer.alloc();
byte[] buffer = b.getBufferData();
int offset = b.getOffset();
// Place data into buffer starting at offset
b.setLength(buffer.length);
getChannel(ps).sendPacket(b, dest);

Figure 5: Example code to source a packet from an AA

public void receivePacket(InChannel c, NetBuffer b,
    Atrib a) throws IOException {
    getChannel(ps).sendPacket(b, a.local_addr, a);
}

Figure 6: Example code to forward a packet from an AA

each VNet-hop and the receivePacket method will be invoked. Note that the
packet will not necessarily be intercepted at each IP hop, but rather every VNet
hop in the VNet topology. After processing the packet at an intermediate node,
the AA typically forwards the packet towards the destination. Figure 6 shows
an example of how to forward a received packet. Notice that in addition to
providing the destination address, the entire Atrib object is also supplied as
the third argument. This provides the necessary ancillary information to ensure
that the network headers of the forwarded packet resemble those of the received
packet.

The final example shows how to reply to a packet. Figure 7 is a code fragment
that receives a packet and echoes the packet contents back to the sender. In this
case, the destination address supplied is the source address of the packet. Note
that no attribute parameter is required in this case.

5 Putting it all together: a complete Ping AA

In this section, we provide a complete Ping AA using some of the code exam-
pies in the previous sections. The functionality of the Ping AA is to test VNet
connectivity between the local EE and a remote EE. The Ping AA tests con-

public void receivePacket(InChannel c, NetBuffer b,
    Atrib a) throws IOException {
    getChannel(ps).sendPacket(b, a.remote_addr);
}

Figure 7: Example code to reply to a packet from an AA
nectivity by traversing the network in a hop-by-hop manner to the destination, then reversing and returning hop-by-hop to the source. It also measures the time to complete this round trip. The reason for using hop-by-hop is for exposition purposes, and in particular, so that we can show how packets are forwarded at intermediate hops.

Like every application for the ASP EE, the Ping application code comprises two parts: the UA and the AA. The UA executes on the local machine and outside the ASP EE environment. The UA provides an interface between the user (either a human or a regular non-active application) and the active-network (through the ASP EE). The UA is responsible for launching the Ping AA and waiting for the returning Ping packet. The Ping AA, on the other hand, is the code that is invoked by each ASP EE along the path.

The term Ping packet is different to the term Ping AA. The Ping AA refers to the program (i.e. the class byte-code) that is loaded by the ASP EE. On the other hand, the Ping packet is the datagram that is transmitted between EEs. The datagram includes an ASP EE header (as well as IP, UDP and possibly an ANEP header). Also included in the Ping packet is an AA-specific payload of 17 bytes.

5.1 Ping AA

The complete source for the Ping AA is given in Figure 8. As mentioned in section 4, the AA must extend the AAContext class and in particular must implement the receivePacket method.

The Ping payload contains 17 bytes. It contains a timestamp, an activity ID, and a one byte field indicating direction. The direction byte is used to distinguish between the Ping packet moving in the forward or reverse directions. In particular, the direction byte is 0x00 if the AA is heading in the forward direction, while the byte is 0x01 if the AA has already turned around.

If the packet has been received from the UA then the AA obtains the destination address from the command line arguments. It then sources a packet to that destination address.

When the Ping AA is invoked on the next hop EE, it will arrive with an unknown activity ID. If the destination address is not the local address, the AA will continue to forward the packet towards it’s destination.

The following line determines whether the destination address is local to this node (i.e. an address in the EE’s interface table).

    if (PPI.isLocal(a.local_addr)) {

When the packet has arrived at the destination, it will be echoed back to the sender with the direction byte field set to 0x01.

At the source node, the returning packet will carry a known activity ID. This ID is used to provide a mapping back to the UA that requested the service. After computing the round trip delay, the AA sends a response to the UA and then destroys the current activity.
package apps;

import java.io.IOException, asp.*, asp.net.*, asp.lib.*, apps.lib.*;

public class Ping extends AAContext {
    public void receivePacket(InChannel c, NetBuffer b, Attrib ia) throws IOException {
        String ps = c.getProtocolSpec();
        byte[] buffer = b.getData();
        int offset = b.getOffset();
        AttribNetwork a = (AttribNetwork)ia;
        if (ps.startsWith("api")) {
            Rshlib r = Rshlib.get(c);
            if (b.getLength() == -1) {
                Activity aid = (Activity) r.getObject();
                if (aid != null) {
                    r.close();
                    aid.destroy();
                }
                return;
            }
            String args[] = r.argv(b);
            if (args == null) {
                r.send("ignoring extra input");
                return;
            }
            AddressNet dest;
            if ((args.length != 1) || (!dest = AddrUtil.String2AddressNet(args[0]))) {
                r.send("usage: Ping address");
                r.close();
                return;
            }
            Activity aid = Activity.create();
            aid.setObject(r);
            r.setObject(aid);
            Conv.long2byte(buffer, offset, aid.getId());
            Conv.long2byte(buffer, offset + 8, System.currentTimeMillis());
            buffer[offset + 16] = 0;
            b.setLength(17);
            getChannel(ps).sendPacket(b, dest);
            return;
        }
    }

    Figure 8: The Ping AA
Activity aid = Activity.get(Conv.byte2long(buffer, offset));
if (aid == null) {
    if (PPI.isLocal(a.local_addr)) {
        if (buffer[offset + 16] == 1)
            return;
        buffer[offset + 16] = 1;
        getChannel(ps).sendPacket(b, a.remote_addr);
    } else
        getChannel(ps).sendPacket(b, a.local_addr, a);
} else {
    Rhode r = (Rohle) aid.getObject();
    long rtt = System.currentTimeMillis() -
        Conv.byte2long(buffer, offset + 8);
    r.println("RTT = " + rtt + " msecs");
    r.close();
    aid.destroy();
    return;
}

private OutChannel getChannel(String protospec)
    throws IOException {
    OutChannel oc = (OutChannel) getLD(protospec);
    if (oc != null)
        return(oc);
    AttribTransport at = new AttribTransport();
    at.remote_port = PPI.getDefaultDstPort();
    at.local_port = at.remote_port;
    at.tos = AttribTransport.TOS_E2E;
    oc = PPI.openOutChannel(protospec, at);
    putLD(protospec, oc);
    return(oc);
}
A Availability of ASP EE source code

All source and documentation for the ASP EE is available from the project’s web-site http://www.isi.edu/active-signal/ARP/.

References
