The PLAN Tutorial*

for PLAN version 2.2

Pankaj Kakkar with Jonathan Moore and Mike Hicks

November 24, 1997

1 Introduction

The purpose of this document is to provide a short introduction to PLAN and the basic usage of the implementation. It will be of help to people just starting with PLAN, although experienced PLAN programmers might use this as a quick reference.

PLAN is implemented in Java, API 1.1, and is composed of a number of packages with names in the form `PLAN.*`. Basic knowledge needed to execute java classfiles is assumed; please refer to the appropriate Java documentation for more. This document also assumes that installation (of both PLAN and its required components) has completed successfully; refer to the Installation document for that.

In the code examples below, it is assumed that the code is being executed from the `PLAN` directory, created by the distribution. If you would like to execute the code from some other directory, simply prepend the class names given to the `java` command with `PLAN`. For example, rather than typing

```
java ARMain -?
```

you would type

```
java PLAN.ARMain -?
```

This should be useful if you have PLAN installed on a machine shared by a number of users, and you would like to run PLAN programs from your home directory. Note that your `CLASSPATH` must be set up to include the parent directory of the distribution `PLAN` directory, or the code will not work (since the `java` interpreter will not be able to find the appropriate PLAN packages). So, if PLAN is installed in `usr/local/PLAN`, you would update your `CLASSPATH` to include `usr/local`.

*This work was supported by DARPA Contract N66001-96-C-852

2 Uses of PLAN

The PLAN distribution allows you to use PLAN in a number of ways. You can

- set up an Active Router which will receive and interpret PLAN packets
- send a PLAN program to an Active Router to be interpreted
- run a non-network PLAN interpreter in *read-eval-print* mode to test out basic programs

This document will illustrate these three uses.

3 Starting up an Active Router

The first step towards being part of an active network is to set up a local machine as an active router. The "ARMain" class serves as the entry point for the router. For more detailed usage information, please refer to The PLAN Active Router [1] document. Try typing :

```
% java ARMain -?
```

You will get a message about the usage of ARMain :

```
Usage : java ARMain [-i] [-v|-d|-q] [-l logfile] [-ip port]
                    [[-m master] [-h hub] | [-rf routtab file]]
                    [-n n1,n2,...,nk]
```

The `-i` option tells the router to be interactive; this requires you to press enter before sending out any packet. The `-v`, `-q`, `-d` options tell the router to be quiet, verbose, or in debugging mode (extremely verbose), respectively. The `-l` option specifies that output messages should be written to the specified file.

The `-ip` option specifies which UDP and TCP ports the router will use for sending and receiving packets (a default port of 3324 will be used if one is not specified).[1] Using different port numbers allows several PLAN nodes in the same PLAN network to reside on the same machine. Only one Active Router can be using any port at any time.

The `-m` flag allows you to specify a network master node for the Service Layer Routing Protocol (SLRP) [2], and the `-n` flag allows you to request certain other nodes as neighbors. If no master node is specified, it is assumed that you are starting a new master. The `-h` flag may be used to specify an ANON hub to use to link separate PLAN networks. Finally, the `-rf` flag is used to specify a file used by this node to form its routing table. If this option is used, the node may not participate in a SLRP network.

A more detailed desription of all of these arguments may be found in [1].

As an example, let us construct an active network of two nodes on our machine m. Typing in a terminal window of m:

[1] PLAN nodes are identified by a machine name and a port, so the provided port also serves to define part of the active node's identity.

```
% java ARMain -v
23-Nov-97 5:32:12 PM: ARMain: verbose mode on.
23-Nov-97 5:32:12 PM: ActiveRouter.start: Active router up!
23-Nov-97 5:32:12 PM: ActiveRouter: OUT to IPv4UDP : (m, 3324) : fstCookie
23-Nov-97 5:32:12 PM: ActiveRouter: OUT: succeeded
23-Nov-97 5:32:12 PM: SLRPmaster: received a request to add: IPv4UDP : (m, 3324)
23-Nov-97 5:32:13 PM: ActiveRouter: OUT to IPv4UDP : (m, 3324) : newRT
23-Nov-97 5:32:13 PM: ActiveRouter: OUT: succeeded
23-Nov-97 5:32:13 PM: ActiveRouter: IN from IPv4UDP : (m, 3324): fstCookie
23-Nov-97 5:32:13 PM: ActiveRouter: IN from IPv4UDP : (m, 3324): newRT
23-Nov-97 5:32:13 PM: SLRP: received new route table.
```

starts a SLRP master, using the default port number 3324. In another terminal window, if we type:

```
% java ARMain -ip 3325 -m m:3324 -v
23-Nov-97 5:34:17 PM: ARMain: verbose mode on.
23-Nov-97 5:34:18 PM: ActiveRouter.start: Active router up!
23-Nov-97 5:34:18 PM: ActiveRouter: OUT to IPv4UDP : (m, 3324) : addme
23-Nov-97 5:34:18 PM: ActiveRouter: OUT: succeeded
23-Nov-97 5:34:18 PM: ActiveRouter: IN from IPv4UDP : (m, 3325): fstCookie
23-Nov-97 5:34:18 PM: ActiveRouter: IN from IPv4UDP : (m, 3325): newRT
23-Nov-97 5:34:19 PM: SLRP: received new route table.
```

we start another active router, listening on port 3324, specifying the other node that we started as the master.

4 Getting Started

The quintessential first problem solved in any programming language is the "Hello world" program.

4.1 The Hello world program

A PLAN program is a list of definitions, which could be function definitions, exception declarations, or var bindings (in the ML style). In the case of the Hello world program, there is just one function :

```
fun doit () : unit =
    (print (thisHost ()); print(" says : Hello world!\n"))
```

4.2 Run the Hello world program : The PLANStart tool

Before we try and understand the program, let us run it to see if it works!

The PLANStart tool is provided as a means to inject PLAN programs into the network directly from the command line. It is invoked by using the Java interpreter. Typing 'java PLANStart' gives you the usage of PLANStart :

```
% java PLANStart
usage : java PLANStart [-v] [-p port] <code> <RB> <router IPv4 2address>
```

The code argument is the name of the file that contains the PLAN program. The RB argument specifies the initial amount of resource[2] to hand to the packet going into the network. The last argument specifies the host that is to be the entry point into the network.

The `-v` option produces a verbose output. The `-p` option allows you to specify the outgoing TCP port, much as was described above for the active router. This also serves to set the port portion of the identity for the PLANStart program should the `thisHost` local service routine be invoked during the initial invocation (see the ping example below). Note that PLANStart and ARMain should use the same port number, or they will not be able to communicate properly.

Suppose that you typed the Hello world program into a file called `Helloworld.plan`[3], and that you want to use an active host called "MyActiveNode" as the network entry point. Let us also give this program an initial resource amount of 10.[4] Now type :

```
% java PLANStart Helloworld.plan 10 MyActiveNode
```

The program waits for you to give it the next input : the initial invocation. Recall that a PLAN program is a list of definitions - when the program arrives at an active host, you must specify which function to start executing and with what arguments. In the case of the Hello world program, there is only one function to execute, and it takes no parameters, so you type :

```
doit ()
```

Out comes the response :

```
IPv4 : MyActiveHost.domain.name/some.ip.address says : Hello world!
```

Notice that you must press Ctrl-C to get back to the command line. This is because the PLANStart tool, after injecting the program into the network, waits for possible responses. Since it does not know how many responses to expect, it has to wait indefinitely until told explicitly to stop.

As a final note, had you included the `-v` option, you would have seen the following:

```
% java PLANStart Helloworld.plan 10 MyActiveNode
Checking for parse errors in source code ...
Binding Top Environ ...
Getting Initial Resource  ...
Getting Address of Active Router  ...
Parsing the initial invocation ...
```

[2] More on this later

[3] Although PLAN does not impose any restrictions on file names, the .plan suffix is suggested for PLAN programs

[4] We'll see later that the resource bound is irrelevant for this program!

This is useful in debugging errors. At this point you would type the initial invocation, as noted above.

4.3 Understanding the program

This very simple program defines a function called "doit", which takes no parameters, and returns a `unit` result.

The body of the function calls a service routine called "print". The print routine takes a single argument, which is the datum to print. The second print call is invoked on a string, while the first argument is yet another service routine call; the "thisHost" service routine returns a value of type `host` which is a representation of the active router that the program is currently executing on.

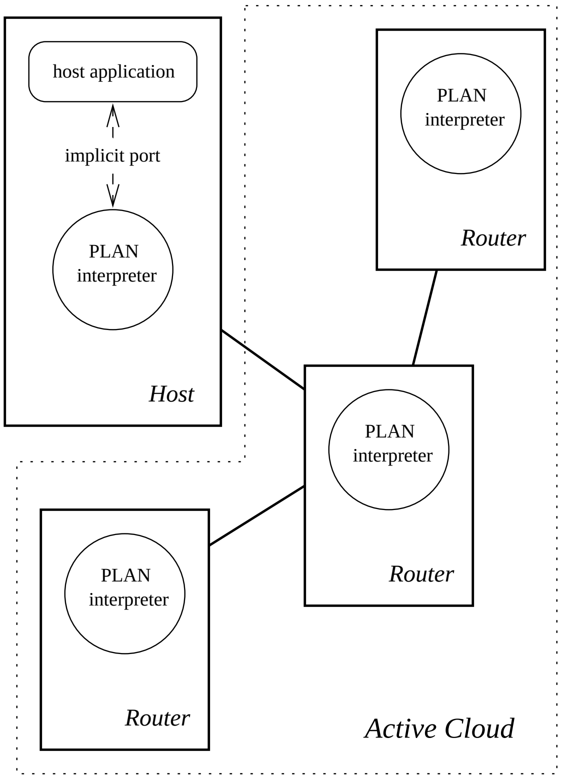

Figure 1: The PLAN network environment

To illustrate how the print routine is working, let us briefly look at how PLAN programs are *injected* by a host into the active network, and how the host obtains their output. Figure 1 illustrates a hypothetical active network and a host with an application that desires to inject a PLAN program into that network. To do this, the application opens a connection to the local PLAN interpreter, and constructs and submits a PLAN packet via this connection. The PLAN packet contains the PLAN code to execute, which in our example is the "Hello World" code, as well some other things that we need not worry about yet. This connection also serves to pass output from the PLAN program back to the host. It is via this connection that PLANStart receives the output generated by the print command, and it is PLANStart that displays this output on your screen.

In PLAN, this connection is referred to as a *port*. Ports are duplex connections between host applications and active routers. The active router side of the port always receives PLAN packets, while the application side receives output in a number of formats, depending on the service command used; for the print routine, the output format is a string. When a PLAN packet is first injected into the network onto a particular node, it is associated with its injection port on that node. That way, output routines such as print have a default port upon which to send their output. These routines will only work on the injection node, because that is where the port resides. For example, in Figure 1, the PLAN program in the packet submitted by `myhost` will have its print output sent back to `myhost` via the injection port so long as it is executing on `myhost`. If the program were to move to another node and then execute the print routine, an exception would be raised.

For a precise description of what the various service routines do, and a greater exposition on ports, please refer to the PLAN Programmer's Guide [3].

5 Moving on

Now that we know and are familiar with the PLANStart tool and the basic format of a PLAN program, we can write more useful programs. PLAN is meant specifically for writing programs to be used in an active network - so let us try and run a very simple program that actually uses the active network.

5.1 The Ping program

The Ping program tries to reach a remote host, and once that task is accomplished, it tries again to reach the original source. Finally, it prints "Success" at the original source.

```
fun ping (source:host, destination:host, outgoing:bool) : unit =
  if outgoing and (thisHost () = destination) then
    OnRemote (ping (destination, source, false),
              source, getRB (), defaultRoute)
  else
    if not outgoing and (thisHost () = destination) then
      print ("Success")
    else OnRemote (ping (source, destination, outgoing),
                   destination, getRB (), defaultRoute)
```

5.2 Understanding the program : the OnRemote primitive

The **OnRemote** primitive is one of the two primitives provided in the language for specifying computation at a node across the network. It takes four arguments:

What The first argument specifies what to execute at the remote node. It must be of the form of a function call. Note that the arguments to the function call will be evaluated on the machine invoking OnRemote, while the function will be evaluated on the remote machine.

Where The second argument specifies where to execute the packet next at. Note that the packet will go through (possibly) many other active nodes in the process of getting to the specified host, but it will not be executed at any intermediate nodes.

How much The third argument specifies how much of the current packet's resources to give to the outgoing packet.

How to The fourth argument specifies which routing function to use to get to the remote host.

The other network primitive provided is **OnNeighbor**. It requires that the second argument be on the same (AN virtual) link-layer as the current host, and hence does not require the fourth argument.

The program consists of just one function definition. This function, called "ping", takes three arguments and returns a unit result. The first two arguments are of type `host`, and mean the source and destination[5] of the packet for a particular leg of its journey. The third argument is a boolean value which specifies whether the packet is outgoing or otherwise.

The body of the function is divided into three cases by the 'if .. then .. else' constructs.

- If the packet is outgoing and it has reached its destination, it is time to turn back. The program then uses the OnRemote primitive. It specifies that the next execution should be at the original source. The function to be executed is ping itself, but with the source and destination switched (since this leg of the journey goes from the destination to the source), apart from outgoing being set to false. It also gives all of its current resources to the outgoing packet, and specifies "defaultRoute" as the routing function to use to get to the source.

- If the packet is incoming and it has reached its destination, it has reached the remote host *and* made its way back to the original source - the packet then just prints a message.

[5] Note that these are independent of the packet header

- If none of the above cases were true, the packet is still on the source and about to begin the first leg of the journey. The OnRemote call sends it on its way.

5.3 Run this program

We'll use the PLANStart tool again, and we'll use the active network we set up at the end of Section 3, so our two nodes are m:3324 and m:3325. Suppose that you want to ping m:3325 from m:3324. Also suppose that the ping program is contained in a file called 'ping.plan'. We'll give this program a resource bound of 60 (will be explained later).

Then you invoke the program by the following :

```
% java PLANStart ping.plan 60 m:3324
```

And the initial invocation would be :

```
ping (getHostByName ("m:3324"), getHostByName ("m:3325"), true)
```

Recall the initial invocation for the Hello World program – simply the function call `doit()`. Here, the initial invocation is a bit more complex. We first specify that we want to execute the `ping` function when we reach our injection node, and then we specify the arguments to give to that function. Here, the first two arguments are themselves function calls. The interesting thing is that these arguments are evaluated *locally* before being sent to the injection point. Therefore, the `getHostByName` function is executed locally, on the injecting host in order to resolve hostnames into PLAN values of type `host`. It is these PLAN values that are actually sent in the PLAN packet. `getHostByName` is one of a number of *local services* that are made available to programs using PLANStart. These local service calls may only appear, and will only be evaluated, in the initial invocation provided to PLANStart. For a complete list of local services, please refer to the PLAN Programmer's Guide [3].

After typing this in, you should see a message on your screen :

```
Success
```

6 The resource bound

6.1 Rationale

Consider the following PLAN program :

```
fun ping_pong(pingHost:host, pongHost:host) : unit =
    OnRemote (ping_pong (pongHost, pingHost),
              pongHost, getRB (), defaultRoute)
```

A very simple program to write and run, but if this program were to be left loose on the network (without any resource bound control), it would keep shuttling between two hosts and eat up costly bandwidth.

6.2 Managing the resource bound

The concept of a resource bound is provided as a solution to this problem. Each packet has associated with it a certain amount of "resources". The packet can only access this value through the getRB () service routine; it can specify the amount of resource bound it wants to give out of its own bank to an outgoing packet through the third argument of OnRemote and OnNeighbor.

Each hop that a packet traverses decreases its resource bound by one. Additionally, whenever a packet causes a new packet to be sent out into the network (through OnRemote or OnNeighbor) its resource bound is decremented by the amount it allocated to the outgoing packet.

If at any point the packet does not have enough resource bound left, either to traverse the next hop or to give out to an outgoing packet, the "NotEnoughRB" exception is raised, which can be handled by the program. No more packets may be sent when the resource bound reaches 0.

6.3 The two examples and resource bound

In the first example that we saw, there were no network primitives, no packets going out. The resource bound was totally irrelevant in that context.

In the second example, the program always gave all of its current resource bound to an outgoing packet. Between the local and remote hosts, the packet might traverse a number of hops, and the resource bound will be decremented before each hop. The initial amount of resource bound given to the packet must be enough to take it to the remote host and get it back. In the example, we assumed that the packet would not take more than 30 hops on each leg, so we gave it a total of 60 units of resource bound initially.

7 The Go tool

In addition to PLANStart, the distribution also provides a tool for running your programs in a read-eval-print loop. Type :

```
java interpreter.Go <code file>
```

to load definitions from the specified PLAN source code file. You will be then prompted with a `-->` and you can respond with a valid PLAN expression to compute, followed by a semi-colon. For example:

```
java interpreter.Go Helloworld.plan
--> 1 + 1;
2
--> doit();
IPv4 : MyActiveHost.domain.name/some.ip.address says : Hello world!
--> "hello "^"world";
"hello world"
-->
```

The network primitives are not available in this tool; if an expression evaluation tries to use OnRemote or OnNeighbor the evaluation is aborted. Also, the Router Services (as described in the PLAN Programmer's Guide [3]) are not available on the read-eval-print loop.

Press Ctrl-C to exit the loop.

8 Conclusion

That ends the short tutorial. For more details on the actual semantics and specs, refer to the PLAN Programmer's Guide [3].

References

[1] M.W. Hicks, "The PLAN Active Router," PLAN documentation, October 1997.

[2] J.T. Moore with M.W. Hicks, "A Service Layer Routing Protocol for PLAN," PLAN documentation, September 1997.

[3] J.T. Moore and M.W. Hicks, "PLAN Programmer's Guide," PLAN documentation, 1997.