PLAN Service Programmer's Guide*

for PLAN version 3.2

Michael W. Hicks

July 25, 2001

1 Introduction

This document is meant to be a guide to programmers intending to write services to be made available to PLAN programs. The document describes the basics of the active router implementation, and the preferred paradigm for extending it with new services and host applications.

The current PLAN implementation is written in OCaml [1] version 2.02, while earlier versions were written in Java [3], and have their own documentation.

2 The PLAN Implementation

A PLAN node consists of a number of distinct components: the *basis*, the *interpreter*, the *network* and *link layer* interfaces, and the *services*.

The basis forms the core information about PLAN needed by both the active router and host applications. This includes definitions of PLAN packets, values and types (all in file `basis/basis.ml`), and how these entities are marshalled to be sent across the wire (file `basis/marshalling.ml`).

The interpreter is the part of the active router that actually evaluates the PLAN programs that it receives. Each PLAN program is compiled to an AST (defined in `basis/basis.ml`) which is evaluated at the evaluation destination. The interpreter is implemented in the file `interpreter/eval.ml`, which makes direct use of the basis and network functions. Core services are defined in the file `interpreter/svc_impl.ml`.

The network interface defines the concept of the network available to the PLAN interpreter (implemented in file `planet/net.ml`). It is this interface that is used by the interpreter to transmit packets during the evaluation of the *OnRemote* and *OnNeighbor* primitives, as well to transmit values to a host application via the use of PLAN ports (with help from `planet/ports.ml`).

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

To properly send a packet, the network interface makes use of address resolution, defined in file `arp/arp.ml`. This implements a PLAN version of the standard IP address resolution protocol (ARP).

The address resolution interface makes use of different underlying linklayer technologies, all of which are assumed to have broadcast semantics. The entry point to each is the file `net/linklayer.ml`, which provides a uniform interface on top of each of the different link layers. Each link layer implements the signature in file `net/linklayer_impl.ml`; there are currently two link layers: Ethernet and IP (in `net/ethernet_impl_file.ml` and `net/ip_impl_file.ml`, respectively). Also defined here are the concept of a network layer address (in `net/activehost.ml`), and DNS resolution (`net/hostfile.ml`).

Security services are defined in the `security` directory. Both services for authentication and authorization are provided. A thorough explanation of these services and some of their implementation details are presented in [5].

The services define the bulk of the functionality of the active router, and are amply described in the PLAN Programmer's Guide [4]. Each service package is implemented in its own directory.

Finally, when the plan daemon is built with the active loader, it uses code in the `loader` directory. Files which are common to both the loader and the standard pland are found in the `common` directory. In order to provide certain security guarantees, the loader makes use of module thinning, meaning that the interface seen by an externally loaded module presented by the running program may be different than the interface available to compiled modules. The specification of alternate interfaces (and sometimes implementations) is automated by the `builder` program, whose source is in the `builder` directory. It takes the `builder/safeloader.build` file as its default specification file (as indicated in the Makefile).

3 Services in PLAN

Writing services for PLAN is a two step process. First, the OCaml code that is to constitute the functionality of the services is written. This will make use of files from the PLAN implementation, so the implementation must be available in order to compile the new service code. Secondly, the services must be installed in the PLAN router. This may be done in two ways: either statically or dynamically. Installing services dynamically requires a slightly different implementation than the static way, so we shall first look at how to make static services.

3.1 An Example: A Random Number Generator

Before going into detail about how to write services in general, we shall demonstrate how to add a new service with an example. The new service, *rand*, returns a randomly-generated integer. The code presented in these examples may be found in the plan/switchlets directory of the distribution; other examples are also present in files in this directory.

The first thing to do is to write the OCaml code that implements the service. This should be in its own file; you want to make sure that only the functions that should be visible to other services and/or PLAN programs are visible in the file's interface. The second thing to do is to write wrappers for each of these functions which conform to what is expected by PLAN. These will also go in their own file. For our random service we first write the file rand.ml, which makes use of OCaml's Random module:

```
(* initialize the generator with the current time *)
let _ = Random.init (truncate (Unix.time ()))

(* the actual random function *)
let rand () =
  Random.int 0x3fffffff
```

The interface for this file shall only export the rand function, so it will look like this (in file rand.mli):

```
(* initialize the generator with the current time *)
val rand : unit -> int
```

The next thing to do is to write the proper wrapper for the rand function so that it may be called by the PLAN interpreter. All service functions should have the following type:

```
val service: Basis.active_packet * (Basis.value list) -> Basis.value
```

where service is the name of the function to be exported to PLAN. Let's look at the rest of the prototype.

value is an algebraic (aka structural) type that defines the form of all PLAN values. Since a PLAN value should be returned by all services invoked from a PLAN program, all service functions should return type value. In addition, it is possible that a service could raise a PLAN exception or cause an execution error. Both of these possibilities are handled by raising OCaml exceptions of type PLANException (defined in file interpreter/planExn.ml), and ExecException (defined in file interpreter/eval.ml), respectively.

The arguments to the service function are a list of value's, and an active_packet. When this function is invoked as a service call from PLAN, each element of the value list shall correspond to the PLAN value that was provided as an actual argument. The active_packet parameter shall be bound to the invoking PLAN packet, and may contain information of interest to the service, such as the currently available resource bound of the packet.

Applying this API we write the file rand_svc_impl.ml:

```
open Basis
open Eval
```

```
open Services

let rand (p,l) =
  match l with
    [] ->
      Int (Rand.rand ())
  | _ -> typecheck_args "rand" l []; Unit
```

For this service, if everything proceeds correctly, a PLAN value which corresponds to a PLAN integer is returned. The current version of PLAN requires that services do their own checking—this is for efficiency purposes. In general, a service routine will match against the expected form of its arguments in the first case of the pattern match, and then call `typecheck_args` for the default case. `typecheck_args` matches the actual arguments with a type specfication having type `Basis.plan_type list`. In this case, no arguments are expected, so the specification is empty (`[]`). `typecheck_args` will return `true` if typechecking succeeds, and raise an `ExecException` otherwise. In this case, we expect typechecking to always fail (since a type-correct program would have matched the initial case), and thus to raise an exception.[1]

Finally, we must add a `register_svcs` function to the module to install its new services. This method serves to place the name of each function corresponding to a PLAN service in the interpreter service symbol table. We can also provide the PLAN type for this function, which allows the static typechecker to infer the type of the service without requiring ¡tt¿svc¡/tt¿ declarations in the PLAN packets. Here is the final version of the code:

```
open Basis
open Eval
open Services

let rand (p,l) =
  match l with
    [] ->
      Int (Rand.rand ())
  | _ -> typecheck_args "rand" l []; Unit

let register_svcs () =
  register_svc("rand",rand,Some "void -> int")
```

The `register_svcs` routine simply maps the string defining the name of the PLAN service, rand, to the function that implements that service, `rand`. This registration routine should be the only one that is available to other services (since the actual functionality in the `rand` module is already available), so its interface is (in file `rand_svc_impl.mli`):

[1] In previous versions of PLAN, the service writer basically had to inline `typecheck_args` for each service. This new arrangement should not hurt performance in the common case, and simplify things for the implementor.

```
val register_svcs : unit -> unit
```

Static installation

To statically install this service, we have to add it to the Makefile to be compiled. Let's assume that all of the files we've defined (rand.ml, rand.mli, rand_svc_impl.ml, rand_svc_impl.mli) are in the directory rand inside of the plan directory itself. First find the defintion of the variable BASE_DIRS:

```
BASE_DIRS = basis interpreter planet port util RIP resident arp csum
  frag net reliable dns exp flow security crypto builder
```

This definition should be extended to include the name of your service directory, in this case rand:

```
BASE_DIRS = basis interpreter planet port util RIP resident arp csum
  frag net reliable dns exp flow security crypto builder rand
```

A little further down, you will find the comment:

```
# The various packages to be installed
```

Just below it is a list of definitions (in alphabetical order) of the various logical file groupings. We shall add a group RAND to this list:

```
RAND = rand.ml rand_svc_impl.ml
```

A little further down, we add the definition:

```
SA_RAND=$(addprefix rand/,$(RAND))
```

Finally, we extend the definition for the variable STANDALONE_OBJS to include SA_RAND:

```
STANDALONE_OBJS=$(SA_COMMON) $(SA_UTIL) $(SA_CRYPTO) $(SA_NET) $(SA_BASIS) \
  $(SA_ARP) $(SA_PLANET) $(SA_PARSER) $(SA_RELIABLE) $(SA_INTERP) \
  $(SA_PLANET_SVC) $(SA_ARP_SVC) $(SA_PORT) $(SA_RIP) $(SA_FLOW) \
  $(SA_CSUM) $(SA_REL_SVC) $(SA_DNS_SVC) $(SA_QCM) $(SA_KEYNOTE) \
  $(SA_SECURITY) $(SA_RESIDENT) $(SA_FRAG) $(SA_EXP_SVC) \
  $(SA_RAND) custom.cmo router.cmo daemon.cmo
```

The files should now be properly compiled when you type make, and will be linked into the system. One detail remains: we must call the register_svcs function in rand_svc_impl.ml once the node has started in order to make the rand function available to PLAN programs. This is done by editing the file custom.ml. Currently the file consists of (in part) the following code:

```
let customize (portnum,sockfile,keyfile',
       neighbors,rout_file,host_file,policy_file,authlist) =
  let keyfile =
```

...

You would change this to indicate that the rand service should be registered:

```
let customize (portnum,sockfile,keyfile',
       neighbors,rout_file,host_file,policy_file,authlist) =
  Rand_svc_impl.register_svcs ();
  let keyfile =
```

...

Recompile and you should now have your new service.

Dynamic installation

The default PLAN daemon does not allow for dynamic service installation. However, there is provided with the distribution an executable called `plan_loader` which is augmented to allow dynamic installation of services. To build this executable, do `make loader`. This will build `plan_loader` and place it in the `bin` directory.

To build a loadable version of each of your files, you will need to do the following:

- at the top of each `.ml` file, add the line
  `open Safeloader`
- at the bottom of the `rand_svc_impl.ml` file, add the line
  `let _ = register_svcs ();;`

Additionally, some functions which are available to services statically are not necessarily available to ones dynamically linked. This is the case with the `Random.init` function. The reason for this is safety, but the policy (which functions to hide or not hide) is a matter of current research. Thus, the modified files will look like the following.
rand.ml:

```
open Safeloader

(* initialize the generator with the current time *)
(* let _ = Random.init (truncate (Unix.time ())) *)

(* the actual random function *)
let rand () =
  Random.int 0x3ffffff
```

rand_svc_impl.ml:

```
open Safeloader
open Basis
open Eval
open Services
```

```
let rand (p,l) =
  match l with
    [] ->
      Int (Rand.rand ())
  | _ -> typecheck_args "rand" l []; Unit

let register_svcs () =
  register_svc("rand",rand)

let _ = register_svcs ();;
```

The next thing to do is to compile these files. The easiest thing to do is copy the two `.ml` files to the `switchlets` directory. Then:

```
[ switchlets ]> make rand.cmo
Makefile:11: .depend: No such file or directory
ocamldep -I ../loader *.mli *.ml > .depend
ocamlc -I ../loader -thread -nopervasives -c rand.ml -o rand.cmo
[ switchlets ]> make rand_svc_impl.cmo
ocamlc -I ../loader -thread -nopervasives -c rand_svc_impl.ml -o rand_svc_impl.cmo
[ switchlets ]>
```

The two resulting `.cmo` files, `rand.cmo` and `rand_svc_impl.cmo` may now be dynamically loaded, using the `installServices` service function. In the `rout_tests` directory is a PLAN program that puts a wrapper around this service function to indicate success or failure. Assuming there is a router running locally, here is how you would install the service, using `inject`:

```
[ plan ]> inject rout_tests/install.plan 1
install("Rand",getBlobFromFile("switchlets/rand.cmo"))
Install succeeded

[ plan ]> inject rout_tests/install.plan 1
install("Rand",getBlobFromFile("switchlets/rand_svc_impl.cmo"))
Install succeeded

[ plan ]>
```

Note that the string `"Rand"` is ignored (an artifact of earlier implementations). If the install were to fail, the message "Install failed!" would be printed.

Now that the service is loaded, we can invoke it from PLAN, using the following plan function (in the file, say, rand.plan):

```
fun foo() : unit =
  print(rand())
```

Here is the invocation:

```
[ plan ]> inject rand.plan 1
foo()
164667826

[ plan ]>
```

For more details about how services are installed, please refer to the PLAN Programmer's Guide [4].

3.2 Writing Service Code

To properly write OCaml code that is to be made available as services in PLAN, a number of rules must be followed.

- **Mutual Exclusion**
  There are no primitives in PLAN to allow for mutual exclusion. Therefore, services must implement their own to guarantee that PLAN programs do not cause inconsistent state. This is especially important when implementing routing functions which may be altering a table while a PLAN program is extracting values from it.

- **Form of Services**
  A new service for PLAN should have the following form. It will consist of N `.cmo` files which make up the code for the new service. The OCaml source that constitute these services must be compiled with access to the PLAN implementation. This is most easily done by putting the files in the `switchlets` directory, and then making them with `make` *file*`.cmo`.

  One of these files should contain a function that will install services into the interpreter symbol table, using the `register_svc` function (in the file `interpreter/services.ml`). Service variables and exceptions may be installed as well, using `register_value` and `register_exn`, respectively.

  All functions that are installed in the symbol table should have the signature

  ```
  val service: Basis.active_packet * (Basis.value list) -> Basis.value
  ```

  where `service` is the name of the function you wish to make available.

  Each exported function is responsible for its own verification (arity, argument type-checking). If verification fails, some form of `ExecException` should be thrown. As described above, this can typically be handled by `Eval.typecheck_args`. If some other problem occurs with the execution of the function, then a `PLANException` should be thrown, which will be visible to the executing PLAN program. For example, if the PLAN exception *BadHost* is to be raised, then the OCaml code would execute

  ```
  raise (PLANException("BadHost"))
  ```

All PLAN exceptions that are thrown by services must also be installed in the symbol table by the `register_exn` function or else PLAN programs will not be able to handle them.

- **Dynamic vs. Static Loading**

  To install services statically requires two steps:

  - First create a separate directory for your service. All source files should be in this directory, and it should be placed in the plan directory. In other words, the service `foo` will have its source in the directory `plan/foo`. The Makefile of in the plan directory should be altered to include this new code.
  - Second, the `custom.ml` file must be altered to invoke any initialization code of your service.

  To install services dynamically involves using a PLAN program to transfer your files to the appropriate node and install them. This is done by using `inject`'s `getBlobFromFile` service to read the files from disk into a PLAN packet to be injected. This packet travels to the node to receive the new code and then invokes `installServices` (see the PLAN Programmer's Guide [4] for more). If services depend on each other (for example, in our example `rand_svc_impl.ml` depends on `rand.ml`), they must be loaded in the proper order, or dynamic linking will fail.

  Note that loaded modules persist until the active router is halted. This is because OCaml does not currently garbage collect modules. Newly loaded modules can shadow old ones, but will not eliminate them.

4 Writing Host Applications

A host application is an application that lives on an end-node and communicates with PLAN active nodes via PLAN ports. PLAN ports can be opened from a host application by opening a connection on the port that the PLAN router is listening on.[2] Also, a PLAN port may be opened from an active router using the *openPort* service. Once the connection is established, it is assumed that all messages going to an active node are marshalled objects of type `active_packet`. Messages coming from the active node are service dependent. For example, the *print* service sends strings, while the *deliver* service sends marshalled objects of type `value`. Marshalling is done via the `marhsalling` module. All of these classes are present in the PLAN `basis` directory.

An example host application, called `server`, can be found in the `port` directory. It listens for connections on a specified port, and prints out `deliver`ed values. It may be used as a template for your own applications.

[2]These connections are, by default, implemented using Unix domain sockets. A port number n is translated into the file `/tmp/swaren`. They may also be implemented as TCP sockets by compiling with option `-DPLANPORT_TCP`. This is to allow applications to be written in other languages on different platforms, e.g. Java on Windows 98.

To facilitate the implementation of host applications in Java, we have written some stub code that facilitates the transmission of PLAN values in Java. The stubs are present in the `java` directory; see the `README` file there for more details.

References

[1] X. Leroy. "The Objective Caml system, documentation and user's guide," `http://pauillac.inria.fr/ocaml/htmlman`.

[2] D. de Rauglaudre. "Camlp4, a Preprocessor for Object Caml," `http://pauillac.inria.fr/camlp4/index.html`.

[3] K. Arnold and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.

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

[5] M.W. Hicks, "PLAN Security Guide," PLAN documentation.