# PLAN Programmer's Guide*

for PLAN version 3.2

Michael W. Hicks, Jonathan T. Moore, and Pankaj Kakkar

July 25, 2001

## 1 Introduction

PLAN version 3.2 is implemented in OCaml, version 2.02. While this document attempts to be as implementation-independent as possible, certain aspects are implementation dependent, and will refer to OCaml as necessary. An earlier, Java-based version of PLAN (2.2) is also available and has its own documentation suite.

This document should serve as a basic guide and reference to PLAN programming. Please see the PLAN grammar [6] for syntactic details, our design paper [3] for rationale, and [5] for details on the Active Network we have built using PLAN, called PLANet.

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

# Contents

## 2 PLAN

PLAN is a simple, functional programming language based on the simply-typed lambda calculus with some added primitives to express remote evaluation. PLAN may be thought of as an Active Network scripting language such that each active packet consists of a PLAN program. PLAN programs may invoke *services*, which make up a library of functions resident on an Active node. Services are described in more detail in Section 7.

PLAN is resource limited in two ways. All PLAN programs are guaranteed to terminate, limiting the resources consumed on any particular active node. Further, PLAN applications (that is, series of PLAN packets spawned from an initial packet) are limited by a global resource bound, thus limiting the resources consumed from the network at large.

The newest version of PLAN supports static type inference of source programs, eliminating (some of) the need for type annotations. In order for the inference algorithm to properly check calls to the services, the programmer must provide *service declarations* in his program indicating the types of service functions or values. PLAN functions and bindings may be polymorphic, as well as those of services. More on these may be found in Section 7.

## 3 PLAN packets

To be evaluated, a PLAN program is bundled into a packet, injected into the active network, and then evaluated. As a result of evaluation, the packet may create other packets which may then be evaluated at other nodes in the network.

### 3.1 Terminology

A PLAN *chunk* consists of three components: the *code*, the *fn_to_exec*, and the *bindings*; the latter two are referred to collectively as the *invocation*. The code consists of a series of definitions which bind names to abstractions; these abstractions may be *functions*, *values*, or *exceptions*. The invocation consists of the function call (i.e., function name *fn_to_exec* and actual parameters $a_1, \ldots, a_n$ – the *bindings*) to be evaluated at the next evaluation point, where the set of all definitions in the code part (as well as the predefined service functions) serve as the legal environment for the call. Note that the *bindings* must be *values* since PLAN supports a *call-by-value* semantics. A *chunk* is both conceptual and concrete: PLAN values of type `'a chunk` are first-class *chunk*s.

A PLAN *interface* identifies a link to a physical network. Interfaces are represented by values of the PLAN type `dev`. Each interface has both a network and a link layer address (the form of the former is universal while the latter is physical network-specific). Since a PLAN node might have links to several physical networks (a router, for example, in the current Internet, has several interfaces each of which may have a different IP and Ethernet address), a single node might be associated with several PLAN interfaces.

A PLAN *packet* carries a PLAN chunk to be executed on an active node. In addition to the chunk, the packet also contains some additional fields: the *evaluation destination*, the *resource bound* (explained later), and the *routing function* (among others). The evaluation destination (or *evalDest*) names the next active node which should evaluate the chunk. The routing function serves to define how the packet will be transported from the current node to the evalDest. The resource bound (or *RB*), indicates the sum total of the number of hops this packet or any of its progeny may make. This serves to limit the network resource usage of any PLAN application.

Finally, a PLAN *application* consists of a series of PLAN packets that comprise a task. This is different than a *host* application, which is a program that runs on a end-node in the network, and may be written in some language other than PLAN. In the simplest case, a PLAN application is defined by the injected packet and all of its resulting progeny. More often, PLAN applications may cooperate with host applications to complete a task.

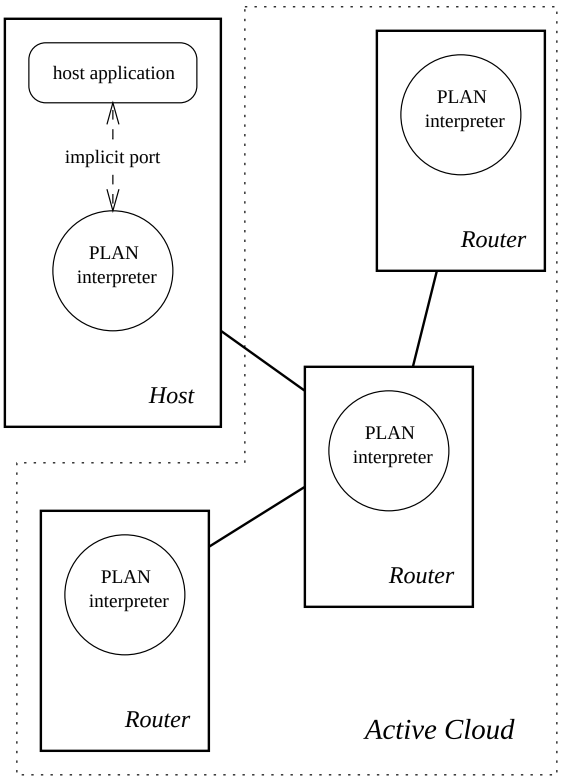


Figure 1: *PLAN network*: the PLAN evaluation environment

### 3.2 Model of Evaluation

To understand how PLAN programs evaluate, it is imperative to understand their evaluation environment, the PLAN network, depicted in Figure 1. A PLAN network consists of hosts, which form the end-points of the environment, and routers, which form the cloud. Collectively, hosts and routers are referred to as nodes. Hosts are essentially distinguished from routers in that they may serve host applications which form part of a PLAN application. All nodes must be capable of evaluating PLAN programs.

A PLAN packet begins its life at a host, specifically within a host application. The application constructs a PLAN packet and *injects* it into the active network by handing it to the PLAN interpreter on its host. This is done by creating a *PLAN port* between the application and the interpreter, called the injection PLAN port (or implicit PLAN port). This PLAN port allows the interpreter to provide output to the application, and allows the application to submit new PLAN packets.

The local interpreter then proceeds to route the packet to its evalDest, making use of the provided routFun to determine the next hop. At each hop on the way to the evalDest, resource bound is subtracted from the packet; if at any time the resource bound is exhausted, the packet is terminated and a message is sent back to the source for potential recovery. When a PLAN packet reaches its evalDest, its program is evaluated. First, the code is parsed top-down to register the top-level bindings. All function bindings respect *static scoping*; that is, the set of bindings available to a function when it is evaluated are those that were available when it was defined. Evaluation then begins with the function call $f(a_1, \ldots, a_n)$ defined by the invocation. Any value that would have been returned by the function is discarded.

Any exceptions (see Section 4) which are not explicitly handled by the PLAN program will be processed by the packet's designated *handler*; see Section 4.2.

# 4 Exceptions and error handling

### 4.1 Exceptions

There are two syntactic constructs dealing with exceptions:

- `try` $e_{\text{try}}$ `handle` $\epsilon$ `=>` $e_{\text{handle}}$
  Execute the expression $e_{\text{try}}$. If $\epsilon$ is an exception literal, then if the exception $\epsilon$ is raised and not caught during the execution of $e_{\text{try}}$, expression $e_{\text{handle}}$ is evaluated. If $\epsilon$ is not a literal exception, it is viewed as a variable name to which any exception raised during the evaluation of $e_{\text{try}}$ will be bound, for an ensuing evaluation of $e_{\text{handle}}$.

  Using the variable-binding form of the `try`...`handle` block allows any PLAN exception to be caught. Therefore a simple, in-packet error-reporting system may be set up. For example :

```
svc print : 'a -> unit

fun exnreport(h:host list,e:exn) =
  (print("I raised ");
   print(e);
   print(" on ");
   print(h))

svc thisHost : void -> host list
svc getRB : void -> int
svc defaultRoute : host -> host * dev

fun main(home,...) =
  try
    ...
  handle e =>
    OnRemote(|exnreport|(thisHost(),e),home,getRB(),defaultRoute)
```

- `raise` $\epsilon$
  Raise the exception $\epsilon$. Evaluate the handling expression of the tightest lexically-enclosing `try`...`handle` construct.

Note that service calls and language constructs may raise exceptions which can be caught with `try`...`handle` blocks. Language level exception provided are:

- *ServiceNotPresent* is raised when a service call is attempted for which the service is not resident on the active node. Note that because service calls are syntactically identical to function calls, the evaluator assumes that *any* unbound function name in an application is a service, thus raising this exception.
- *DivByZero* is raised upon an attempted division by zero.
- *Hd* is raised when `hd` is applied to an empty list.
- *Tl* is raised when `tl` is applied to an empty list.
- *NotEnoughRB* is raised when the current packet runs out of resource bound, or if it attempts to allocate to a child packet more resource bound than it has available.
- *HostNotLocal* is raised when a router is asked to forward a packet to a node which is not one of its link-layer connections.
- *NoRouteEntry* is raised when the invoked routing function does not know of a route for the specified destination node.

## 4.2 Error handling

Any exception raised by a PLAN program can be caught by a top level `try`...`handle` block. However, in the case that an exception arises outside of packet execution (such as if the packet runs out of resource bound before it can be evaluated), or if no enclosing `try`...`handle` block is defined, the packet *handler* is used for recovery. This is the name of a function, stored as part of the packet, which should be invoked back on the source. This function must have the following signature:

$$string \times host \times \alpha chunk \rightarrow unit$$

The arguments are, respectively, the *error* which occurred, the *node* on which it occurred, and the (truncated) *chunk* which was (or would be, on arrival to the evalDest) evaluating when the error occurred. These arguments have the following properties:

- the *error* argument will either be the name of a PLAN exception, or one of the following:[1]

```
ParseExn.UndefinedType
ParseExn.SyntaxError
ExecExn.TypeError
ExecExn.ArgMismatch
ExecExn.AppError
ExecExn.UnboundVariable
ExecExn.BadRoutFun
ExecExn.EmptySequence
ExecExn.Generic
Unix.Unix_error
Net.MarshallingException
Unknown
```

- the *chunk* argument is truncated to ensure that the error can be properly sent back to the source without need for fragmentation. Specifically:
  - The *code* field of the chunk is omitted
  - The *fn_to_exec* field of the chunk is left as is
  - The *bindings* to the *fn_to_exec* are truncated as follows:
    Values of type `unit`, `int`, `char`, `bool`, `host`, `port`, `key`, and `dev` are left as is. Values of `string`, `exception`, `blob` are truncated to the first 16 characters/bytes. Lists are truncated to the first element only (which is itself truncated). Tuples retain their arity while the values they contain are truncated. Finally, a chunk has its code and bindings omitted, and the *fn_to_exec* is truncated to at most 16 characters.

[1] The mapping of runtime error to one of these strings is performed by the function `exn_to_string` in the file `router.ml`.

A default handler, `defHandler` is provided. There also exists a PLAN service, `abort`, which produces behavior similar to that of an invoked handler. Both are described in detail in Section 7.

# 5 Some PLAN Primops

Here we describe some of the more interesting PLAN primitive operations (namely, network and list operations).

## 5.1 Network Primitives

The network primitives are the most interesting aspect of PLAN, as they enable mobile computation via generating and sending new active packets.

- *OnRemote*
  The first basic network primitive, *OnRemote*, is quite general. Its syntax is:

  `OnRemote(` $E$, $H$, $Rb$, $Routing$ `)`

  The meaning of this primitive is: evaluate $E$ on node $H$. Furthermore, use the *Routing* function to determine how to get to $H$. Finally, the current packet donates *Rb* of its resource bound to the child packet.

  More specific restrictions follow:

  - $E$ must be an expression of type $\alpha\,chunk$. Typically, a chunk literal is created used the syntax described in Section 6.1. This syntax resembles a function call in which the function $f$ to be called is surrounded by $|$'s. The arguments of the function $(a_1, \ldots, a_n)$ are evaluated locally (becoming $v_1, \ldots, v_n$), while the function itself will be evaluated remotely.
  - $H$ is an expression of type *host*.
  - *Rb* is an integer indicating how much of the parent's global resource bound should be transferred to the child packet.
  - *Routing* should be a service (not a PLAN function) of type $host \rightarrow host \times dev$. $Routing(h)$ should return some node $h'$ which is a neighbor of the current node (with the idea being that $h'$ is the next hop on a route to $h$ in the routing scheme represented by *Routing*).

  On success, the call to *OnRemote* will create a PLAN packet which is sent to $h'$, decrementing the current packet's resource bound by the designated amount. The invocation part of the PLAN program in the newly created PLAN packet consists of $f(v_1, \ldots, v_n)$ (where $f$ is the function named in the chunk, and $v_1, \ldots, v_n$ are the actual paremeters) and the code part consists of the subset of the code of the invoking program that is required to properly evaluate the invocation. The remainder of the packet is defined by the remaining three arguments to *OnRemote*. Within the currently executing program *OnRemote* returns *unit*.

In the event of an error, one of the following exceptions will be raised (and the creation of the packet aborted):

 - *NotEnoughRB* will be raised if *Rb* exceeds the parent packet's current resource bound.
 - *HostNotLocal* will be raised if the *Routing* function maps $H$ to a node which is not a neighbor of the current node.

- *RetransOnRemote*
  The second network primitive, *RetransOnRemote* is similar to OnRemote, except it tries to ensure *reliable* delivery of the PLAN packet. The syntax is :

  `RetransOnRemote` ( $Cs, K, n, H, Rb, Routing$ )

  Here, we want the chunks $Cs$ (which could be the fragmented parts of a PLAN packet) to be sent to the node $H$ using the routing function *Routing*, and resource bound *Rb* for each chunk. The packet should be retransmitted every second until an acknowledgement is received (as indicated by the invocation of the service function `reliableAck`, see below), up to a maximum of $n$ times. Specifically :

   - $Cs$ is a list of chunks.
   - $K$ is an expression of type *key*, and "names" the chunks $Cs$. In particular, the retransmission of these chunks will be halted when the function `reliableAck` (see Section 7.2.9) is executed on the sending node with $K$ as an argument.
   - $n$ is the maximum number of times to retransmit the chunks $Cs$.
   - $H$ is the node at which the chunks $Cs$ should be evaluated.
   - *Rb* is the amount of resource bound that is given to *each* chunk in $Cs$.
   - *Routing* is the routing function used to route the packets to $H$.

  *RetransOnRemote* will return *unit* on success, and might raise one of the following exceptions in case of an error :

   - *NotEnoughRB* will be raised if *Rb* exceeds the parent packet's current resource bound.
   - *HostNotLocal* will be raised if the *Routing* function maps $H$ to a node which is not a neighbor of the current node.

  For an example of the use of *RetransOnRemote*, see Section 7.2.9.

- *OnNeighbor*
  The final primitive, *OnNeighbor* is similar to *OnRemote*, with the restriction that the generated child packet must execute on a neighbor of the current node. The syntax is:

  $$\texttt{OnNeighbor(}\ E,\ H,\ Rb,\ D\ \texttt{)}$$

  Here, we want $E$ to be evaluated on $H$ (a neighbor of the current node), and $Rb$ of the current packet's resource bound will be transferred to the child packet. $D$ specifies the *device* that the packet should be sent through. More specifically:

  - As with *OnRemote*, $E$ must be a chunk.
  - $H$ is a expression of type *host*, and must be a neighbor (that is, on the same link layer network) of the current node.
  - $Rb$ is an integer expression indicating how much of the parent's resource bound should be transferred to the child packet.
  - $D$ is an expression of type *dev* that indicates the device that the packet should be sent through. A device value corresponds to an entity specified in the interface configuration file. See the active router manual for more details [4]. Devices are normally acquired as the result of calling a routing function (like `defaultRoute`), or by querying how the packet arrives (`getSrcDev`)—see Section 7.1 for details on these services.

  On success, the call to *OnNeighbor* will return *unit*. In the event of an error, one of the following exceptions will be raised:

  - *NotEnoughRB* will be raised if $Rb$ exceeds the parent packet's current resource bound.
  - *HostNotLocal* will be raised if $H$ is not a neighbor of the current node.

### 5.2 List Iterators

The list iterators are based upon the well-known functional programming construct *fold*, with left-associative and right-associative varieties provided. Since PLAN does not (currently) support language-level parametric polymorphism or higher-order functions[2] the list iterators are provided as language primitives, rather than as proper functions.

- $foldl : (\alpha \times \beta \to \alpha) \times \alpha \times \beta\ list \to \alpha$
  Although the type above is correct, current PLAN programs may only have a function name appearing in the function argument to *foldl* (which

[2] Some services, such as `print`, do support these features.

is only natural, since PLAN does not have anonymous functions). The meaning of

$$\text{foldl}(f, a, [b_1; \ldots; b_n])$$

is

$$f(f \ldots f(f(a, b_1), b_2) \ldots, b_n)$$

- $foldr : (\alpha \times \beta \rightarrow \beta) \times \alpha\ list \times \beta \rightarrow \beta$
  As for *foldl*, the function argument to *foldr* must be a function name. The meaning of

  $$\text{foldr}(f, [a_1; \ldots; a_n], b)$$

  is

  $$f(a_1, f(a_2, \ldots f(a_n, b) \ldots))$$

### 5.3 Summary of Other Operators

| | |
|---|---|
| Unary Operators | |
| `~` | integer negation |
| `not` | boolean negation |
| `hd` | return the first element of a list (*car*) |
| `tl` | return the list without its first element (*cdr*) |
| `fst` | return the first element of a tuple |
| `snd` | return the second element of a tuple |
| `#`$n$ | return the $n$th element of a tuple |
| Binary Operators | |
| / | integer division |
| % | modulo |
| * | integer multiplication |
| + | integer addition |
| − | integer subtraction |
| `and` | boolean conjunction |
| `or` | boolean disjunction |
| < | integer less than |
| <= | integer less than or equal to |
| > | integer greater than |
| >= | integer greater than or equal to |
| = | equality |
| <> | inequality |
| :: | cons |
| << | left shift arithmetic |
| >> | right shift arithmetic |
| `xori` | bitwise xor |
| `andi` | bitwise and |
| `ori` | bitwise or |
| `noti` | bitwise negation |
| `explode` | convert a string to a list of characters |
| `implode` | convert a list of characters into a string |
| `ord` | return the ASCII value of a character |
| `chr` | return the character with the given ASCII value |
| `^` | string concatenation |

## 6 Layering

PLAN provides an interesting method of implementing traditional protocol encapsulation.

## 6.1 Chunks

Essentially, a PLAN computation be made first class by turning it into a *chunk* (as described in Section 3.1). A PLAN value of type $\alpha$ `chunk` is specified by surrounding a function's name with $|$'s at the call site (the $\alpha$ part is determined by the return type of the given function). This newly created chunk can in turn be given as an argument to another function, which itself can be made into a chunk. This function should perform some processing of its own, and then as its last action evaluates the original chunk. This process mimics *encapsulation*. When the lastly created chunk is sent to the destination and evaluated, each encapsulated chunk is evaluated, the reverse order in which they were made. This process mimics *demultiplexing*. Examples of how layering is used may be found in Sections 7.2.9 and 7.2.8.

When a chunk is created, a static analysis is performed to determine how much of the code of the invoking packet should be bundled into the newly created chunk. This analysis is conservative in that it will include at least the needed code, and perhaps more. For example, consider the following PLAN program, which implements *ping*:

```
svc thisHostIs : host -> bool
svc getRB : void -> int
svc defaultRoute : host -> host * dev

fun ping (source, destination) =
  if (thisHostIs (destination)) then
    OnRemote (|ack| (), source, getRB (), defaultRoute)
  else
    OnRemote (|ping| (source, destination),
              destination, getRB (), defaultRoute)

svc print : 'a -> unit

fun ack() = print ("Success")
```

When this packet is sent to *ping* a remote destination, the `ping` function will be specified in the initial invocation, therefore at least the code for ping is required. The analysis then looks through the definition of ping for any other chunk creations. It then finds the first one, `ack`. Because the code for `ack` may be needed later, when `ping` is actually invoked, the definition for `ack` is retained as well.

Here is the conservative part. As the analysis recursively searches the program through its chunk creations, it keeps track of the *lowest* function definition, and retains all code from the start of the definitions up to that point. This is why the analysis does not have to look at normal function invocations: normal static scoping, which enforces a "define before use" model, will take care of correctness there.

Any unneeded code will be *pruned*. Because of the conservatism of the algorithm, the order of the definitions in the code matters. For example, consider the creation of the `ack` chunk above at the destination. Because `ack` is the second function defintion, the code for the `ping` function will also be retained, even though it isn't needed (`ack` will never call `ping`). However, if we reorder the defintions so that `ack` comes first, the `ping` code will be properly discarded. A better flow analysis could be less conservative.

# 7 Services

Much of the functionality of PLAN comes from its services. While the initial release of PLAN offered only a smattering of services, the current version provides a rich library from which to choose. Services may be divided into two categories: the *core* services, which are present on all Active Routers, and additional service *packages*. Each service package consists of one or more service routines made available to PLAN programs.

As with all functions called from PLAN, service functions may take multiple arguments (and not a single argument which may be a tuple, as in ML); this is presented below as a comma-separated list of types. And, like all PLAN functions, services always return a value; a service will return *unit* in the case that the output has no meaning.

Because service routines are not guaranteed to be ubiquitious, the PLAN programmer must declare the types he expects the services to take in his PLAN program using *service declarations*. These may be present at the top-level and have the form `svc` *id* `:` *argtypes* `->` *resulttype*. Types may be polymorphic; type variables are specified with a `'` followed by a letter. Here are some example declarations:

```
svc defaultRoute : host -> host * dev
svc fragment : ('a chunk,int) -> unit chunk list
svc getRB : void -> int
```

The first names the `defaultRoute` function, which takes a single argument of type `host` and returns a pair of type `host` $\times$ `dev`. The second names the `fragment` function which takes two arguments, of types $\alpha$ `chunk` and `int`, returning a list of `unit chunks`. Finally, `getRB` takes no parameters and returns a value of type `int`.

## 7.1 Core Services

These are the services which the programmer may expect to be present at every active router. As such, service declarations for these services need not be supplied for typechecking purposes.

- $getRB : () \to int$
  Returns the current amout of resource bound held by the packet.

- $getSource : () \rightarrow host$
  Returns the node specified in the packet's source field. Assuming that nobody has tampered with the packet on the way to the current node, this is the node that the packet's oldest ancestor (which may be the packet itself) originated from.

- $getSrc : () \rightarrow host$
  Same as $getSource$.

- $getSessionKey : unit \rightarrow key$
  Each packet has a unique key, called the *Session key* associated with it. The service returns this key, which may be used in various services that take key arguments.

- $thisHost : () \rightarrow hostlist$
  Returns a list of all network addresses for devices on the current node.

- $thisHostOf : dev \rightarrow host$
  Returns the network address which corresponds to the given device. If the argument is not a proper device, the exception *InvalidDevice* would be raised.

- $thisHostIs : host \rightarrow bool$
  Returns *true* if the given address matches any of the addresses of interfaces on the current node.

- $gettimeofday : () \rightarrow int \times int$
  Return the current time since 00:00:00 GMT, Jan. 1, 1970, as a (seconds, microseconds) pair.

- $timeStamp : () \rightarrow string$
  Returns the current time of day as a string.

- $length :\ \alpha\ list \rightarrow int$
  Returns the length of the given list.

- $member : (\alpha,\ \alpha\ list) \rightarrow bool$
  Checks whether the given element is in the list or not.

- $remove : (\alpha,\ \alpha\ list) \rightarrow \alpha\ list$
  Removes the specified element from the given list.

- $print : \alpha \rightarrow unit$
  If this call occurs on the node on which this PLAN application originated, then the value will be converted to a string (see `toString`, below) and the output will be made available to the initiating process. The underlying mechanism for print is the PLAN port created by the injecting host program between it and the originating node. PLAN ports are discussed in more detail in Section 7.2.2.

- $getSrcDev : () \rightarrow dev$
  Returns the interface on which the packet arrived.

- $getDevs : () \rightarrow dev\ list$
  Returns a list of all interfaces on the current node.

- $getMTU : dev \rightarrow int$
  Returns the **M**aximum **T**ransmission **U**nit for the physical network that the given device connects to.

- $toString : \alpha \rightarrow string$
  Converts the argument (a PLAN value) into a string.

- $staticRoute : host \rightarrow host \times dev$
  PLAN provides a static routing system, based on local routing tables read in from files at startup. This service returns the nexthop and interface from a specified destination according to the static routing table. The format of the static routing table file is given in the file `FILE_FORMATS` in the `plan` directory.

- $getStaticRoutes : unit \rightarrow (activehost \times activehost \times dev) list$
  Prints the static routing table as a list of three tuples; the first element of the tuple is the destination, and the next two elements are the next hop given as the host and the device to send through to get there. The default route will have destination *any_addr*.

- $getHostByName : string \rightarrow host$
  Takes as input a *name*, according to the following grammar:

  $$\begin{array}{lcl} \textit{name} & ::= & \textit{host} \mid \textit{host} : \textit{port-num} \\ \textit{host} & ::= & \textit{ip-addr} \mid \textit{domain-name} \\ \textit{port-num} & ::= & \textit{int-literal} \\ \textit{domain-name} & ::= & \textit{string-literal} \end{array}$$

  where *ip-addr* has the form $n.n.n.n$ where each $n$ is an integer from 0 to 255. If the *port-num* argument is not provided, then it is assumed to be the implicit PLAN port number of the invoking node. *getHostByName* attempts to resolve this *name* into a value of the PLAN *host* type. This mapping is currently determined by the PLAN DNS, which obtains mappings from a file indicated at node startup time. Raises an $UnknownHost$ exception on failure.

- $defaultRoute : host \rightarrow host \times dev$
  The default routing function. By the default, this uses the RIP routing function (see below).

- $getNeighbors : () \rightarrow (host \times dev)\ list$
  Using the default routing function's table, this function determines the "neighbors" of the current node; that is, those nodes that are on one of the physical networks to which the current node is attached.

- $eval : \alpha\ chunk \rightarrow \alpha$
  Subtracts 1 from the current packet's resource bound, and evaluates the given chunk as long as some resource bound remains; if not an *OutOfRB* exception is raised. The environment is first reinitialized to be the default for the node. The result of evaluating the chunk is returned from `eval`. Notice that you could indirectly simulate recursion using this service (which must terminate due to loss of resource bound):

  ```
  fun f (x:int):int =
      ...
      eval (|f| (x-1))
      ...
  ```

- $evalBlob : blob \rightarrow unit$
  Same as `eval`, except the argument is of type *blob*. The argument is first cast to a chunk, which results in a type error on failure. The return value is always discarded; hence the return type is always `unit`.

- $zeroRB : () \rightarrow unit$
  Sets the resource bound of the packet to zero. This is used for error packets (for example, the ones sent by the `abort` service routine) to ensure that error reporting packets do not spawn off any new packets.

- $abort : \alpha\ chunk \rightarrow unit$
  This service routine can be used by the packet to stop executing on detection of unforeseen problems and send an error report to the source. On a call to *abort*, execution of the packet is terminated, and a special packet is created and sent to the source. On arrival at the source, this packet sets its resource bound to zero, and evaluates the given chunk (discarding the result).

  This error reporting packet is given a resource bound of 32. Thus even if a packet has run out of resource bound (and thus cannot even send an error report back normally) it would be able to report the problem to the source.

- $defHandler : (string, host, \alpha\ chunk) \rightarrow unit$
  This is a default error handling service, which prints to the console of the source node the error caused by a PLAN program which was not explicitly dealt with. See Section 4.2 for details on error handlers in general.

- $queryGC : () \rightarrow int \times int \times int \times int \times int \times int \times int \times int \times int \times int \times int \times int \times int \times int$
  Returns a tuple containing statistics from the garbage collector. These are, respectively:

  - Number of words allocated in the minor heap.

- Number of words allocated in the minor heap that survived a minor collection and were moved to the major heap.
- Number of words allocated in the major heap, including the promoted words.
- Number of minor collections.
- Number of major collections.
- Total number of words in the major heap.
- Number of times the major heap size was increased.
- Number of words of live data in the major heap, including the header words.
- Number of live objects in the major heap.
- Number of words in the free list.
- Number of objects in the free list.
- Size (in words) of the largest object in the free list.
- Number of wasted words due to fragmentation. These are 1-words free blocks placed between two live objects. They cannot be inserted in the free list, thus they are not available for allocation.
- Number of heap compactions.

• *logGC* : () → *unit*
  Causes GC stats to be printed out on the console of the current node.

• *broadcast_addr* : *host* is a value representing a broadcast address. Packets sent to this address will be received by all neighbors of the current node.

• *any_addr* : *host* is a network address representing all interfaces on the current node.

• *loopback_addr* : *host* represents the address of the current node; packets sent to this address will be received by the current node.

## 7.2 Service Packages

There are ten service packages provided with the current PLAN distribution: *install*, *port*, *resident*, *RIP*, *arp*, *flow*, *dns*, *auth*, *frag*, *reliable*, and *csum*. The latter three may be used to implement traditional protocol layering in PLAN. They are described below.

### 7.2.1 install

This package provides routines that allow service packages to be dynamically installed. It is however only available in a specialized version of PLAN, available in the distribution as an executable called `plan_loader`. For more on this, refer to [2].

- $installServices : (string \times blob)\ list \rightarrow unit$

  This executes the Caml bytecode in the *blob* on the invoking node. Each element of the input list is a pair whose first element is a string which is ignored (it is included for backward compatibility), while the second element is the bytecode file containing code for the service package to be installed. Elements of this list are installed in order starting from the head first, and therefore the order is important. All packages being installed will be dynamically linked into the router's environment.

  If a failure occurs during this process, a diagnostic will be printed to the router console, and the exception *BadService* will be raised.

  An obvious side-effect of this service is that loaded packages will consume memory on the active router, a fact that could be maliciously or inadvertently exploited. Therefore, we anticipate that future releases of PLAN will require authentication before allowing a PLAN program to dynamically load services.

#### 7.2.2 port

This package provides a means for PLAN programs to interact with host applications. The package makes use of the PLAN ports, which have PLAN type `port`. A PLAN port is a duplex connection between an active router and a host application. The active router side of the PLAN port always receives PLAN packets, while the application side receives output in a number of formats, depending on the service used. When a PLAN packet is first injected into the network onto a particular node, it is associated with its injection PLAN port on that node. This PLAN port is referred to as the *implicit PLAN port* of a packet.

The module in the source distribution which implements PLAN ports is `planport.ml`. Here is its signature:

```
type plan_port
exception ClosedByPeer
exception PortOpFailed
exception PortClosed
exception NoPacket

(* client side ports *)
val open_port : int -> plan_port
val close_port : plan_port -> unit
val get_packet : plan_port -> string
val try_get_packet : plan_port -> string option
val timed_get_packet : plan_port -> float -> string
val send_packet : plan_port -> string -> unit

(* server port (only one) *)
val get_local_port : int -> plan_port
```

By default, PLAN ports are implemented as Unix Domain sockets – that is, each PLAN port names a Unix Domain connection between *pland* (the active node daemon) and a host application. Associated with each PLAN port is a *port number*, and the socket opened is called "/tmp/sware" ^port-num. Alternatively, TCP may be used as the underlying connection mechansim; this is done by compiling with the Camlp4 option `-DPLANPORT_TCP`. In this case, the PLAN port number refers to a TCP port number.[3]

PLAN ports may be created implicitly or explicitly. Each Active Router has a server thread listening on a PLAN port whose number is specified when the router is started. Essentially this thread consists of a loop which calls `get_local_port` on the user-specified port number; this routine waits for and accepts client connections. Once a connection is established, another thread is forked which repeatedly calls `get_packet` to obtain client packets and then queues them for processing. The router tracks the connection in a table and creates a PLAN port to name it. Packets that arrive via this connection will have this PLAN port as their implicit PLAN port. The implicit PLAN port can be obtained within a PLAN program with the PLAN routine `getImplicitPort`.

A PLAN port may be created explicitly within an executing PLAN program by the routine `openPort` (which is a wrapper for the `planport.ml` routine, `open_port`. This routine attempts to connect to a server application listening on the specified port number, and if successful returns the PLAN port associated with the connection.

PLAN Ports are only valid on the router which created the PLAN port in the first place. In other words, if PLAN port $P$ lives on router $R$, but a PLAN program attempts to send data through it while executing on router $S$, the operation will fail and raise an exception.

Note that PLAN port names are not stored in any explicit way on the routers. Therefore, a PLAN program that wishes to use a PLAN port must retain the value of type `port` which names the PLAN port. This serves to prevent one PLAN application from using the PLAN ports of another. Note that PLAN programs may use the resident data facility to store PLAN port values on the routers (see Section 7.2.3 for more on resident data).

A Unix Domain (or TCP/IP) socket connection named by a PLAN port is closed whenever a PLAN program explicitly closes it via *closePort*, or when the peer terminates the connection.

- $getImplicitPort : () \rightarrow port$
  This function takes no arguments and returns the implicit PLAN port of the invoking PLAN program. Note that this PLAN port will not be valid if it was previously closed using the *closePort* service.

- $openPort : int \rightarrow port$
  This function attempts to create a connection between the invoking host and a server application residing on that host listening on the specified

[3] PLAN 2.2, which was implemented in Java, used TCP/IP. Using TCP as the underlying connection mechanism should enable Java and Windows compatibility.

Unix domain (or TCP/IP) socket. If successful, the function returns a PLAN port naming that connection. On failure, the exception *OpenFailed* is raised.

- *closePort* : *port* → *unit*
  This function closes the connection associated with the given PLAN port. If the PLAN port was closed previously, or does not reside on this router, the exception *InvalidPort* will be raised.

- *printPort* : (*port*, $\alpha$) → *unit*
  This function is identical to the core service *print* except that the output is transmitted via the specified PLAN port, rather than via the invoking packet's implicit PLAN port. The commands `print(a)` and `printPort(getImplicitPort(),a)` are identical. If the PLAN port was closed previously, or does not reside on this router, the exception *InvalidPort* will be raised.

- *deliver* : (*port*, $\alpha$) → *unit*
  This function is identical to *printPort*, above, except in the form of its output. Rather than converting the specified PLAN value to a string, the value is instead structurally *marshalled* to be sent across the connection. This marshalling scheme is not currently documented, but is defined by the class `basis/marshalling.ml`. This routine is useful when the receiving host application wishes to manipulate PLAN values directly. This application, if written in OCaml, would make use of the `basis` package, which defines OCaml versions of PLAN's values (the ones used by the implementation, in fact). An example of such an application is provided with the PLAN distribution, called `server`. This application listens on the specified PLAN port for connections, and then prints any PLAN values it receives through the PLAN port. A sample PLAN program to use with this application is `rout_tests/deliver.plan`.

  If the specified PLAN port was closed previously, or does not reside on this router, the exception *InvalidPort* will be raised. If delivery fails for any other reason, the exception *DeliveryFailed* will be raised.

### 7.2.3 resident

Resident data is data that remains on the router after the PLAN program that created it has terminated. This feature was added to PLAN to enable programs to easily perform network traversals, such as depth first search (DFS) and breadth first search (BFS). To perform these searches, it is necessary to have some way of *marking* nodes so they are not revisited, as the network topology may be cyclic. Rather than providing a special-purpose service to enable node marking, we opted to provide the more general service of resident data. An example of the use of resident data to perform a DFS is given in the example program `rout_tests/traceNet.plan`.

Resident data is stored in a table on the router which is indexed by a string which names the data, and a session *key.* This key serves to differentiate data stored by different instances of the same PLAN application.

One hazard of allowing resident data is that PLAN programs may (maliciously or inadvertently) leave large chunks of data on a router, consuming memory even after the PLAN program has terminated. To combat this problem, each piece of resident data has a timeout associated with it. This timeout is set by the user when the data is created, but may not exceed 5 minutes (in other words, users may be "nice" and opt to release their data before the 5 minute limit). In the future, we hope to implement a more semantically-appealing way of cleaning up leftover resident data, such as a form of distributed garbage collection. We also anticipate that eventually the use of this service will require authentication.

- $generateKey : () \to key$
  Generates a fresh key which is globally unique.

- $get : (string,\ key) \to \alpha$
  Retrieves the value indexed by the provided string and key. If no such value exists, the exception *NotFound* is raised.

- $put : (string,\ key,\ \alpha\ ,int) \to unit$
  Associates in the table the specified value (the third argument) with the provided string and key. The fourth argument specifies the number of seconds that the data is allowed to live. This value is currently capped at 300.

- $setLT : (string,\ key,\ int,\ int) \to bool$
  Same as `put`, except that before changing the value, a test is performed on the stored data to see if the specifed new value is less than the old stored value, and the value changed only if the comparison returned true. The test and set happens as an atomic operation, i.e., if another PLAN packet tried to look at the data during the call to `setLT`, it would wait until the call completed.

  The result of the comparison is returned.

- $setGT : (string,\ key,\ int,\ int) \to bool$
  Same as `setLT`, except the comparison "greater-than" is used.

- $delete : (string,\ key) \to unit$
  Deletes the value indexed by the provided string and key. If no such value exists, the exception *NotFound* is raised.

### 7.2.4 RIP

This implements a distributed routing protocol based on the standard RIP protocol. The following are included :

- $RIP : host \rightarrow host \times dev$
  Given a node of type `host`, returns a pair consisting of the next hop on the route to the detination and the interface to the next hop. By default, `defaultRoute` is an alias of this function.

- $RIPGetNeighbors : () \rightarrow (host \times dev)\ list$
  Returns a list of all neighbors of the current node. The interface to that neighbor is also returned, along with the node. By default, `getNeighbors` is an alias of this function.

- $RIPRecv : ((host \times int)\ list,\ host) \rightarrow unit$
  Given a table of distances to various nodes from a neighbor specified in the second argument, *RIPRecv* updates the routing table at the current node.

- $RIPGetRoutes : () \rightarrow (host \times host \times dev \times int)\ list$
  Returns the entire routing table at the current node as a list of (*destination, nextHop, interfact, distance*) tuples.

### 7.2.5 arp

This package contains services needed to implement address resolution. This package is based on the standard Address Resolution Protocol (ARP).

- $bind : (blob,\ host,\ dev) \rightarrow unit$
  Registers a binding of the link layer address specified in the first argument to the given node. The third argument is the interface that connects to the node.

  If the binding fails for any reason, the exception $BindFailed$ is raised.

- $retrieveBinding : (host,\ dev) \rightarrow blob$ Returns the link layer address corresponding to the specified node for the given interface. The exception *NoBinding* is raised if a binding is not found.

### 7.2.6 flow

In some applications, it is reasonable to route packets along a route different from the one determined by the default, hop-count based routing protocol. Since hop count metrics tend to approximate latency, applications with higher bandwidth requirements might choose to accept longer latencies via longer paths in order to achieve higher bandwidth. Such an example is described in [5].

This package provides services to help set up and use such alternate routes, called "flows". Associated with a flow is the notion of a "metric," which may be calculated differently for different applications. Each node maintains a value called the "toll" which might be used in determining the metric for a flow. A toll of 1 at each node, with the metric being the sum of all toll values on a route reduces to shortest path routing. Several other strategies can be explored.

A flow is identified using a PLAN `key` (see Section 7.2.3), which must have been generated at the destination.

- $flowSet : (key,\ host,\ int,\ dev) \rightarrow unit$
  This attempts to install a flow route on the current node, given the key associated with the route, the next hop to take from this node for packets moving with the flow, the route metric, and the outgoing interface.

  If there is already a flow route installed associated with the given key, the new route overrides the previous one. It is the PLAN programmer's responsibility to ensure that the metric of the new route is better than that of the previous one, consistent with his definition of the metric.

- $flow : host \rightarrow host \times dev$
  Ignores the argument, and uses packet headers to route.

- $getToll : () \rightarrow int$
  Returns the value of toll at a given node.

- $setToll : int \rightarrow unit$
  Sets the toll value to the specified integer.

### 7.2.7 dns

This package implements domain-name resolution. Mappings are provided by a static table, generated from a file specified at router startup time.

- $getHostByName : string \rightarrow host$
  Takes as input a *name*, according to the following grammar:

$$
\begin{array}{lll}
\textit{name} & ::= & \textit{host} \mid \textit{host} : \textit{port-num} \\
\textit{host} & ::= & \textit{ip-addr} \mid \textit{domain-name} \\
\textit{port-num} & ::= & \textit{int-literal} \\
\textit{domain-name} & ::= & \textit{string-literal}
\end{array}
$$

  where *ip-addr* has the form $n.n.n.n$ where each $n$ is an integer from 0 to 255. If the *port-num* argument is not provided, then it is assumed to be the same PLAN port as the invoking node. *getHostByName* attempts to resolve this *name* into a value of the PLAN *host* type. Raises an $UnknownHost$ exception on failure.

### 7.2.8 frag

This package provides a set of services used to fragment PLAN programs into smaller pieces and then reconstitute them at a remote destination. This service is needed because most physical networks have a maximum deliverable packet size, called the Maximum Transferable Unit, or MTU. Consider the following PLAN program:

```
svc defaultRoute : host -> host * dev

fun sendem (c:unit chunk,x:int*host) =
  (OnRemote(c,snd x,fst x,defaultRoute);x)

svc fragment : ('a chunk,int) -> unit chunk list
svc length : 'a list -> int
svc getRB : void -> int

fun fragment_deliver (dest, path_mtu, c:unit chunk) =
  let val cs = fragment(c,path_mtu)
      val l = length(ds) in
   (foldr(sendem,cs,(getRB()/l,dest)); ())
  end
```

The purpose of the PLAN function `fragment_deliver` is to deliver the chunk `c` to node `dest` and evaluate it there. The problem is that the chunk `c`, when marshalled and encapsulated in a PLAN packet, may be too large to send across one of the links on the path to `dest`. Therefore, `fragment_deliver` first fragments that chunk into $n$ smaller, properly-sized chunks via the call to the `fragment` service. Each of these chunks consists of a call to the service `reassemble`, which takes four arguments: a fragment of the original chunk, the sequence number of the fragment (numbered 0 to $n-1$), a boolean set to *true* for the last fragment and *false* otherwise, and finally a value of type `key` which is uniquely generated and set to the same for each fragment. Next, each fragment is sent via the call to `sendem` from `foldr`. When each fragment arrives at the destination, it calls `reassemble`, which stores each original chunk fragment until all have arrived, at which point it reassembles the original chunk and evaluates it.

Notes:

- The second argument given to the `fragment` service is the size of the packet to be sent, i.e. the MTU. Therefore, the resulting list of chunks will each be slightly smaller than the specified MTU, in order to accommodate the additional overhead of packet encapsulation. In particular, this overhead is the value of the constant `fragment_overhead`.
- Each call to `reassemble` stores the information passed to it in a table, indexed by the `key`. If any fragment is lost, the stored information will be useless and will eventually time out. In particular, by default each fragment may arrive at most *frag_timeout* seconds apart, else the space in the table is freed. By default, this value is set to 10.
- The reassembly table is, of course, of finite size. By default it is set to 211 entries[4]. Any attempt to add an entry to a full table will fail.

[4] A prime number of reasonable size.

- Each packet that calls `reassemble` may have leftover resource bound. This bound is also stored in the table, and then donated to the reconstituted chunk for evaluation.

The following services are included:

- $fragment : (\alpha\ chunk,\ int) \rightarrow unit\ chunk\ list$
  Takes a chunk and a size, and fragments the chunk into chunks of that size minus `fragment_overhead`. Each of the returned chunks contains a fragment of the original chunk. When all of these chunks are evaluated, the original chunk is itself reconstituted and then evaluated.

  The chunk to be fragmented is first marshalled into the form that is actually sent over the wire. It is then divided into small units that can fit into the MTU given, and each of these units is encapsulated as a PLAN value of type *blob*. In case the MTU is too small for any fragment to fit inside it, the PLAN exception MtuTooSmall is raised.

  The list of chunks returned contains one chunk for each blob. Each of these chunks has no code and a call to the reassembly service to be invoked at the remote end.

- $reassemble : (blob,\ int,\ bool,\ key) \rightarrow unit$
  At the evalDest, each fragment puts in a call to the reassembly service, providing the actual packet fragment in the blob, a sequence number to indicate its position in the stream of fragmented packets, a boolean value indicating whether there are fragments with a higher sequence number or not, and lastly a key to identify a particular stream of fragments. Each call to this service merely registers the arrival of fragments, until the last fragment is received, which causes the fragments to be reassembled and the complete packet to be evaluated, or a timeout occurs, which causes the fragments already received to be dropped.

- $fragment\_overhead : int$
  Specifes the additional number of bytes added to a chunk encapsulated by the `fragment` service *plus* standard packet overhead[5]. If overhead is $o$ and MTU is $m$, then useful space in a fragment is only $m - o$.

#### 7.2.9 reliable

This package provides a set of services used in conjunction with the *RetransOnRemote* primitive to ensure reliable delivery of PLAN data. Consider the following PLAN program:

[5]This assumes that the packet which carries the chunk will have routing function `defaultRoute` and handler `defHandler`. If these are not the correct values, the user must adjust the value given to `fragment` by the difference in the number of letters in the names of those functions. For example, if the routing function `RIP` was used, and no handler is specified then the value given to `fragment` could be $mtu - (|"defaultRoute"| + |"defHandler"| - |"RIP"| - |""|)$

```
svc generateKey : void -> key
svc sendAck : ('a chunk,key,int) -> 'a chunk
svc defaultRoute : host -> host * dev
svc fragment : ('a chunk,int) -> unit chunk list
svc getMTU : dev -> int
svc fragment_overhead : int

fun reliable_deliver(dest, c:unit chunk) =
  let val seq = generateKey()
      val d = sendAck(c,seq,10)
      val p = defaultRoute(dest)
      val ds = fragment(d,getMTU(snd p)-fragment_overhead) in
    RetransOnRemote(ds,seq,3,dest,20,defaultRoute)
  end
```

The purpose of the PLAN function `reliable_deliver` is to reliably deliver the chunk `c` to node `dest` and evaluate it there. The program first generates a key to associate with the packets to send; this can be thought of as sequence number which identifies the chunk. More specifically, this key will be used in a call to `reliableAck` on the source to halt further retransmissions of the packets containing `c`. Next, the call to `sendAck` creates a new chunk `d` which "encapsulates" `c` with some additional code to be executed on the destination (the encapsulating code will be shown below); part of the action of `d` is to send a packet to the source to invoke `reliableAck`. Next, `d` is fragmented into a list of chunks `ds` using the fragmentation service (see Section 7.2.8), using the outgoing link MTU for size information. Finally, this list of chunks is sent using `RetransOnRemote`. The list of chunks will be resent at most 3 times (third argument), and each chunk will receive 20 resource bound (fifth argument).

Suppose that the list `ds` consists of two chunks. When the second chunk arrives at the destination, the two chunks will be reconstituted into the chunk `d` which is then evaluated; the code for `d` (created by `sendAck`), is as follows:

```
svc reliableAck : key -> unit
svc defaultRoute : host -> host * dev
svc eval : 'a chunk -> 'a
svc arrived : key -> bool
fun send_ack(c:unit chunk,source,seq,rb) =
  (OnRemote(|reliableAck|(seq),
            source,rb,defaultRoute);
   if arrived(seq) then
     ()
   else
     (eval(c);()))
```

Here, the arguments `c`, `seq`, and `10` which were given to `sendAck` are bound here to `c`, `seq`, and `rb`, respectively (the `source` variable is filled in automatically by `sendAck`). First, a packet is sent back to the source to invoke `reliableAck`; this tells the sender to quit resending. Next, the `arrived` service is invoked with the given key to make sure this isn't a duplicate packet (and if not, it registers the key). If not, the original chunk `c` is finally evaluated (note that the returned value is discarded to satisfy the typechecker).

Some things to note:

- using the link MTU (the `getMTU` call) is not strictly correct; there is no guarantee that the MTU of the local network is $\leq$ the *path MTU* to the final destination `dest`. We have just done so for expositional purposes. See the example program `rout_tests/pathMTU.plan` for a PLAN program which determines path MTU.

- True reliable delivery is not actually acheived here, since chunks are only retransmitted a finite number of times. This is because of the difficulty of managing resource bound. We hope to have a solution to this problem in the near future.

- The example above is of a *reliable packet* which is then fragmented. The alternative is to send *reliable fragments* by reordering the calls to the encapsulation services:

```
svc generateKey : void -> key
svc sendAck : ('a chunk,key,int) -> unit chunk
svc defaultRoute : host -> host * dev

fun sendem (c:unit chunk,x:int*host) =
  let val seq = generateKey()
      val d = sendAck(c,seq,10) in
    (RetransOnRemote([d],seq,3,#2 x,#1 x,defaultRoute);x)
  end

svc fragment : ('a chunk,int) -> unit chunk list
svc getMTU : dev -> int
svc fragment_overhead : int
svc reliable_overhead : int
svc length : 'a list -> int

fun reliable_fragments(dest,c:unit chunk) =
  let val p = defaultRoute(dest)
      val cs = fragment(c,getMTU(snd p)-reliable_overhead)
      val l = length(cs) in
    (foldr(sendem,cs,(getRB()/(l*3),dest)); ())
  end
```

  What happens here is that the chunk `c` is fragmented *first*, and then each fragment is sent reliably with a call to the PLAN function `sendem`. Note that the size of the fragments is set to be the MTU size *minus the overhead* *of encapsulating the reliable code*. Only those fragments which are lost will actually be resent, thus preserving network bandwidth. The tradeoff is that less useful payload is delivered per packet, since each packet sent is encapsulated with both fragmentation and reliability code. In our original example, only the original chunk is encapsulated for reliability, and so this overhead is divided evenly among each of the resulting fragments.

The following services are included:

- $sendAck : (\alpha\ chunk,\ key,\ int) \rightarrow unit\ chunk$
  Takes a chunk and a key, and wraps the chunk into a new chunk which causes an ack to be sent back to the source.

- $reliableAck : key \rightarrow unit$
  Registers the receipt of an ack at the current node for the reliable packet sent out with the given key.

- $arrived : key \rightarrow bool$
  Checks to see if the packet specified by the given key has already arrived.

- $reliable\_overhead : int$
  Specifes the additional number of bytes added to a chunk encapsulated by the `sendAck` service.

#### 7.2.10 csum

This package provides a encapsulation-based checksum service. For more details on workings of encapsulation, see Sections 7.2.9 and 7.2.8. The following services are included:

- $checksum : \alpha\ chunk \rightarrow unit\ chunk$
  The specified chunk is first marshalled into the wire form and a checksum is calculated for it. This wire form is then encapsulated as a PLAN blob, and a new chunk is returned, which has the following as its code part:

  ```
  svc verifyChecksum : (blob,int) -> bool
  svc evalBlob : blob -> unit
  fun unchecksum(b,c) =
    if verifyChecksum(b,c) then evalBlob(b) else ()
  ```

  Where `b` is bound to the wire form of the original chunk and `c` is bound to its checksum. When this code is evaluated at the evalDest, the original chunk will only be executed if its checksum is correct.

- $verifyChecksum : (blob,\ int) \rightarrow bool$
  Verifies if the given blob has the checksum specified as the second argument.

- $checksum\_overhead : int$
  The overhead (in bytes) for the encapsulation described above.

#### 7.2.11 authproto

Each of these services is described in more detail in the PLAN Security Guide.

- $DHmessageOne : (\alpha,\ blob) \rightarrow \alpha$
- $DHmessageTwo : (\alpha,\ blob) \rightarrow \alpha$
- $DHmessageThree : (\alpha,\ blob) \rightarrow \alpha$

#### 7.2.12 security

Each of these services is described in more detail in the PLAN Security Guide.

- $authEval : (\alpha\ chunk,\ blob \times int \times int) \rightarrow \alpha$

### 7.3 Services available at injection

Not all services are available in the arguments to the initial invocation at the point where a PLAN packet is injected into a PLAN network. The services available at the injection point include all the core services, plus the services in the dns package and the following new service:

- $getBlobFromFile : string \rightarrow blob$
  Takes the filename given and reads the binary data into an opaque value of type *blob*. This service could be used to deliver payloads, via PLAN ports, or to install new services via the *installServices* routine. If the specified file can't be found, the *FileNotFound* exception is raised.

### 7.4 Services available at the REPL level

Finally, the plan tool (explained in the tutorial) also limits the availability of services. The ones available are all the core services, plus the services in the csum, frag and reliable modules.

## 8 Batteries Not Included

Some programming language features which are not part of PLAN (but may be included, given that user demand for them exists and that they can be added without compromising safety requirements) are:

- higher-order functions, mutation (i.e., assignment), type declarations, recursive functions, forward function references (i.e. the static scoping enforces a strict "define before use" policy), looping constructs

Some of these are approximated. For example, chunks serve as an approximation for higher-order functions, OnRemote and/or eval may be used to simulate recursion and forward function references, and the resident service package provides more heavyweight, mutable data. In general, the service level may be used to provide functionality not available at the language level.

## References


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

[2] M.W. Hicks "PLAN Service Programmer's Guide," PLAN documentation.

[3] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles, "PLAN: A Packet Language for Active Networks." *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86-93. ACM, September 1998.

[4] M. Hicks "*pland*: The PLAN Active Router," PLAN documentation.

[5] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles, "PLANet: An Active Internetwork." *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124-1133. IEEE, March 1999.

[6] J.T. Moore, "The PLAN 3.2 grammar," PLAN documentation.

[7] M. Odersky and P. Wadler, "Pizza into Java: Translating theory into practice," *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.