Odyssey

Version 1.3 User's Guide

Odyssey Code Group

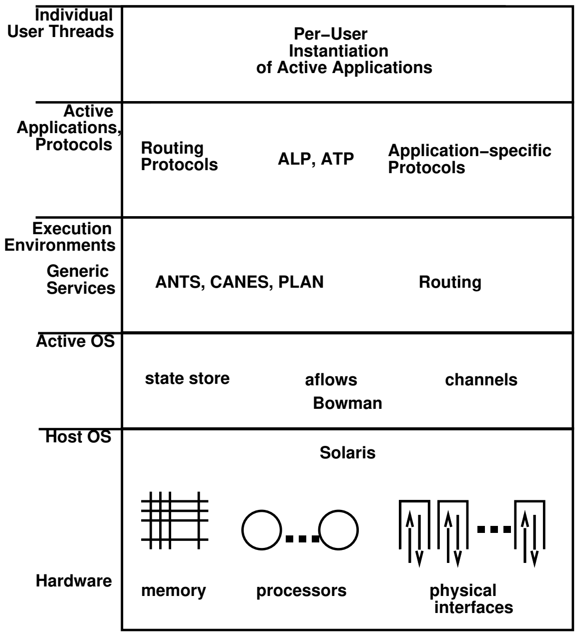
My mind is going...
I can feel it

HAL to Bowman,
2001: A Space Odyssey

1 Odyssey

1.1 Introduction

The Odyssey active networking environment is an implementation of the DARPA active networking architecture *ArchDraft*. It consists of two major components: the Bowman Node Operating System and the CANEs execution environment. The software structure of Odyssey is shown in the Figure. Odyssey assumes a host operating system over which a node operating system is built. Execution environments and generic services that are likely to be useful across environments (e.g. routing) are built using the abstractions provided by the node operating system; user code and particular protocols are run as part of an execution environment.



Odyssey Software Architecture

The Odyssey implementation is multi-thread safe and multi-processor capable and can utilize Solaris Real-Time Scheduling Extensions. The current implementation uses the POSIX threads library and runs as an user-process above `SunOS 5.5.1.` environment in this paper. Its preliminary performance results have been extremely positive.

2 CANEs

2.1 Overview of CANEs

The second major component of Odyssey is an implementation of the CANEs execution environment (as described in *DiAN, RaANP*). The Odyssey implementation of CANEs executes as an aflow within Bowman and provides complete support for underlying and injected programs. Both the underlying and injected programs are demand loaded, dynamically. The slot processing model described in *RaANP* is implemented; both underlying and injected programs can export slots, to which multiple sub-injected programs may be bound. The CANEs implementation also provides mechanisms for variable sharing between programs exporting and sharing slots. The signaling message consists of a computation part (which we ignore here), and a communication part. The communication part of the message identifies the routing schema and topology for the user. This provides information for the CANEs EE to establish the input and output channel plumbing for the user's computation. C++ Linkage for underlying and injected within Odyssey program is now available. The CANEs library itself is included in an extern "C" `{ }` block.

2.2 CANEs Packet API

See Section 3.6 [Packets], page 17

2.3 CANEs Slots

In this section, we describe the slot processing model for active networking. Composition in CANEs is achieved in two steps — first the user selects an "underlying program" that is executed on behalf of the user. The underlying program is offered by the node, and may have provable properties. Depending upon the their needs, users may choose different underlying programs. The users then select/provide a set of "injected programs", which correspond to individual code modules, that can be used to customize the underlying program on a per-user basis. We introduce the notion of a *processing slot*—that localizes the processing of injected code. In effect, slots identify *specific points* in the underlying program where injected code may be executed. Slots are also the mechanism by which the underlying program asserts properties that hold at the node.

Like in an event-driven framework, injected code is "bound" to specific processing slots, and becomes eligible for execution when appropriate slots are "raised". Underlying programs in CANEs define and raise specific slots, and composite services are formed by binding user-specified code to specific slots. More than one injected program may be bound to the same slot— the order of execution of instructions belonging to different injected programs bound to the same slot is non-deterministic. User-supplied code may raise slots during their own execution—this corresponds to recursive events (i.e. events raising their own events). Injected code have an uniform signature, and inter-module communication is achieved via the node store. Forwarding functions would be typical examples of underlying programs. Depending on the user, and the network provider, there may be several forwarding functions available at each node, each of which would define a specific set of processing slots. The underlying program and the composition technique is designed such that we can formally reason about the progress and safety of the resultant computation *on a network-wide basis* independent of the injected code.

2.3.1 Declaring and Exporting Slots

Slots must be declared and exposed to injected programs with the following two macros. The argument `slot_name` for both declarations can be any valid "C" variable name, it is opaque and should not be interpreted in any manner. Its semantics are outside the "C" language. The variable should not be manipulated in any way, and should only be passed as an argument to `canes_raise_slot`. Note that there is no analogous "import_slot" function. Injected programs are "bound" to slots during the canes signaling phase.

```
#define canes_slot(s) _canes_declare_slot(s)
#define canes_export_slot(s) _canes_export_slot(&s, #s)
```

2.3.2 Raising Slots.

This function allows programs which are bound to a slot to execute. All injected programs bound to the slot s will execute concurrently. The parameter `slot_id` should be the same variable as was specified in the corresponding `canes_slot`, and `canes_export_slot` declarations. As a result of preprocessing those macro's, it is now of type `canes_slot_id`.

```
#define canes_raise_slot(s) _canes_raise_slot(s)
```

2.4 CANEs Variables.

An Import/Export mechanism follows which allows variables to be shared with injected programs bound to slots. In this mechanism, the parent is referred to as the underlying or injected program which exports slots, and the children are injected programs which are bound to those slots.

Variables are shared with children ONLY when they are exported by the parent AND imported by the child. No variables of a child are visible to the parent. If more than one injected program is bound to a slot and import a shared variable exposed by the parent, multiple references to the shared variable will exist. Each of these references will be updated synchronously.

2.4.1 Declaring Shared Variables in the Parent.

Parents declare shared variables using the following primitives:

```
#define canes_shared_global(t, v) _canes_shared_global(t,v)
#define canes_shared_global_per_flow(t, v) _canes_shared_global_per_flow(t,v)
```

As an example, to make the `unsigned long current_time` variable shared, it should be replaced by:

⇒`canes_shared_global(unsigned long, current_time);`

or by

⇒`canes_shared_global_per_flow(unsigned long, current_time);`

In the first case, there is a single `current_time` shared variable that will be synchronously updated for each flow as any thread writes into `current_time`. In the latter case, there is a per flow `current_time` variable, and updates to it by one flow do not affect its value in another flow.

The exporting program _must_ declare each shared variable as a global. (The reason being that stack allocated variables are not likely to be around when they are eventually accessed by the injected program) There are cases when global scope is not required — when the exported is sure that all usage will occur when the shared variables are on the stack. There is a plausible (though somewhat cumbersome) scenario (as each time these shared variables are allocated they have to be exported and imported akin to the shmem publish/subscribe.

2.4.2 Declaring Shared Variables in the Child

Children declare shared variables using the following primitives: (The per-flow predicate and the type have to match for reference calls.)

The referencing program may declare shared references within whatever scope is deemed useful. As noted, before each use, however, each reference must be properly initialized using the appropriate canes_import_xxx call.

```
#define canes_referenced_global(t, v) _canes_shared_reference(t, v)
#define canes_referenced_global_per_flow(t, v) _canes_shared_per_flow_reference(t,v)
```

2.4.3 Importing and Exporting Shared Variables

The exporting program **must** declare each shared variable as a global. The reason being that stack allocated variables are not likely to be around when they are eventually accessed by the injected program. There are cases when global scope is not required, when the exported is sure that all usage will occur when the shared variables are on the stack. There is a plausible though somewhat cumbersome scenario: as each time these shared variables are allocated they have to be exported and imported akin to the shmem publish/subscribe. The referencing program may declare shared references within whatever scope is deemed useful. As noted, before each use, however, each reference must be properly initialized using the appropriate `canes_import_xxx` call.

```
canes_boolean_t canes_export_global_per_flow (char *name, canes_generic_t p);
canes_boolean_t canes_export_global (char *name, canes_generic_t p);
```

Polymorphic forms of these functions would have been nice. Each shared variable must be explicitly imported by the injected programs. FWIW, there may be multiple references, each will be updated synchronously, and each reference only uses up some pointer space. Each reference must explicitly import the shared variable before use.

```
canes_generic_t canes_import_global_per_flow (char *name);
canes_generic_t canes_import_global (char *name);
```

2.4.4 Accessing Shared Variables

Each reference to shared variables must be explicitly imported before use. While cumbersome, this will hopefully be beneficial as it requires the programmer of the parent program to **think** about what is being exported. The variables exported are exactly the amount by which the sandbox is **legally** opened by the parent program. Obviously, current non type-safe C allows for any program to wander around in memory and munge any part of core it pleases, but that seems to be hard to check without run-time or static type checking. The following macros, which should be used just as the variable itself would be used, provide import and export functionality:

```
/* access exported variables */
#define c_Ep(v) _canes_access_shared_per_flow(v)
#define c_Eg(v) _canes_access_shared(v)

/* access imported variables */
#define c_Ip(v) _canes_access_referenced_per_flow(v)
#define c_Ig(v) _canes_access_referenced(v)
```

2.5 CANEs Entry

Every underlying program must have a function named `_entry` with the prototype `canes_program_t`. When dynamically linked, this will be the function called by the flow initializer. Both underlying and injected programs have the same entry prototype:

```
#define canes_underlying_program_entry_point "_entry"
typedef void (*canes_program_t)(canes_flow_id_t f);
```

2.6 CANEs GetID

Returns the unique identity of the thread within the CANEs environment. Note that this id is *not* the same as this flow's Bowman aflow id.

```
int canes_getid();
```

2.7 Sending and Receiving CANEs Packets

2.7.1 Sending CANEs Packets

This call sends a packet `pk` of length `len` out the channel `ch_id`, and returns the length of the packet sent. The packet is mutated and freed by a successful canes_send. Therefore packet manipulation or copying of the data within a packet MUST occur before a packet has been sent.

```
canes_packet_length_t canes_send(canes_channel_id_t c, canes_packet_t **p,
 canes_packet_length_t l);
```

Caution: This is a blocking call. A non-blocking version can be made available if need be.

2.7.2 Receiving CANEs Packets

This call blocks waiting for a packet to arrive on the input channel. Upon arrival, the call returns the length of the packet received, `pk` contains the head from the input channel queue.

```
canes_packet_length_t canes_recv(canes_packet_t **p);
```

Caution: This is a blocking call. A non-blocking version can be made available if need be.

2.8 CANEs Timers

2.8.1 Setting a Timer

Timers can be set within any underlying or injected program. The `canes_timeout` function can be called to set the timer. After at least `num_ticks` ticks, function `timer_f(arg)` will becalled unless the timer has been successfully cancelled. The resolution (in ticks) is likely to be near 5000 usecs. `num_ticks` must be given as integer values. The function returns a handle to the timer established which is required to cancel timer.

2.8.2 Prototype for Timeout Handler Functions

This is the function prototype for all timeout handler functions:

```
typedef void (*canes_timer_func_t) (int arg);

canes_timer_handle_t canes_timeout (canes_tick_t t, canes_timer_func_t f,
    int arg);
```

2.8.3 Canceling a Timer

If an existing timer has not gone off, the timer can be canceled using the timer handle `th`.

```
int canes_timeout_cancel (canes_timer_handle_t th);
```

2.9 CANEs Logging

ALL text output should be done with the following `canes_log` function. This will print "format string", variables, in printf style to the canes_output_file iff `b` is true. If `0` is returned, nothing printed.

```
u_int canes_log (canes_boolean_t b, "format_string", variables...);
```

Caution: This is a blocking call. A non-blocking version can be made available if need be.

2.10 CANEs Store

Each user flow is allocated a state store during flow initialization. Datum can be added to this store by the signaling message that starts the flow.

The per-flow store is accessed via the calls

```
canes_boolean_t canes_put (char *k, canes_generic_t v);
```

```
canes_generic_t canes_get (char *k);
```

The `canes_put` call adds a datum to the store. The `key` for the datum is the parameter `k` and the value is the parameter `v`.

The `canes_get` call retreives an existing value from the store. The key is passed as the only parameter. This call will return NULL if the specified key does not exist in the per-flow store.

3 Bowman

3.1 Overview

The Bowman node operating system is built around three key abstractions:

- **Channels.** A Channel is the primary abstraction for communication in Bowman . A channel is a communication end point; it consists of a set of protocols and optionally local and remote addresses. Channels export an interface for reading and writing data onto the channel. In each case, the data is processed according to the channel protocols, i.e., the data is framed with protocol-specific headers upon a channel write and conversely, protocol-specific headers are removed upon reading from a channel.

  The implementation of channels encapsulates all of the protocol processing required to implement individual protocols over each channel. Channels serve to decouple user computations from exactly how the node operating system implements particular communication protocols. Bowman exports interfaces to create, destroy, query and communicate over channels. Details of Bowman channels and the protocols implemented under Odyssey are discussed in Section~\ref sec:channel .

- **A-flows.** An a-flow[1] is the primary abstraction for computation in Bowman . It encapsulates a set of processing contexts and user state. Each a-flow consists of at least one thread and executes on behalf of an identified principal (user of the system). Bowman provides interfaces to create, destroy and run a-flows. Further, multiple threads may be active within a Bowman a-flow — the level of concurrency is user-selected. Execution environments may create a-flows per-user or may just create one a-flow (and not expose its internal per-user state to Bowman ; in such a case, the EE itself is just one user to Bowman .

- **State-store.** The state-store provides a mechanism for a-flows to store and retrieve state that is indexed by a unique key. The Bowman state-store interface provides functions for creating, storing and retrieving data from named state-stores. Further, using the underlying state store mechanism, Bowman also provides an interface for creating indexed registries for arbitrary data types.

Though elegant, the channel, a-flow, and state-store abstractions are not quite enough for a complete implementation of an active node. Interfaces to lower level mechanisms such as memory management, processing threads, synchronization primitives, and device input/ output are required. Thus, Bowman is layered on top of a *host* operating system that provides these lower level services. On the other hand, the Bowman channel, a-flow, and state-store abstractions are too elementary to be useful for most users. Thus, Bowman provides an **extension** mechanism that is analogous to loadable modules in traditional operating systems. Using extensions, the Bowman interface can be extended to provide support for additional abstractions such as queues, routing tables, user protocols and services.

[1] The term "flow" is often used to describe per-user computation state at an active node*NodeOSDraft*. As these flows are, in a manner, *active* —i.e. they refer to both a set of packets and some processing, we adopt the use of the term "a-flow".

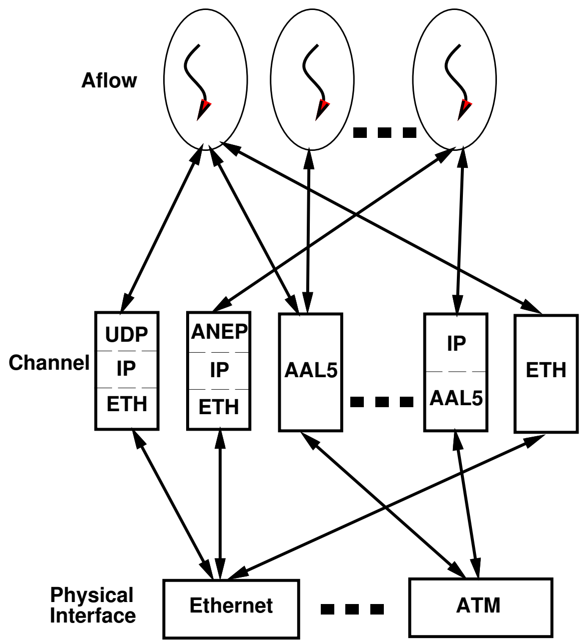
3.2 Channels

3.2.1 Overview of Channel

Channels are an abstraction of the communication resources provided by the node operating system in an active networking environment.

The channels in Bowman are local to an active node — they represent communication *end-points*. Communication between nodes is accomplished via *abstract links*. An abstract link is an association between a set of channels.

Bowman channels are logical entities; i.e., there is a many-to-many mapping between the set of channels and physical interfaces at a node. Obviously, each data operation on a channel resolves to at least one physical interface[2], however, the binding between channels and physical interfaces need not be pre-determined or static. Thus, channels serve to abstract the particulars of available physical interfaces from all higher layers.



Relation between Interfaces, Channels, and Aflows

- **Protocols** Associated with each Bowman channel is a set of protocol functions that encapsulate the processing done as part of data transfer over the channel. It may be helpful to consider channel protocols as a traditional protocol stack, though the underlying implementation need not adhere to this structure. For example, a channel

[2] or a loopback interface.

could be configured with the `{UDP|IP|Ethernet}` protocols much like the Internet protocol stack. Note that there may be an implicit order in the channel protocols, e.g., UDP data must always be encapsulated in IP datagrams. We do not, *a-priori*, enforce any particular ordering. Instead, we tacitly assume that the channel creation will fail if the set of protocols are ill-specified.

Bowman channel protocols need not all be communication related — it is possible to create channels that securely transfer compressed data by specifying, for example, the `{GZip|DES|UDP|IP|Ethernet}` set of protocols[3]. In general, the protocols specified on the peers of a channel should match; however, Bowman channels neither specify nor enforce this condition.

Along with the protocols, all relevant protocol-specific parameters, including peer endpoint addresses, must also be specified when a channel is created. Bowman channels, in general, have both local and remote endpoint addresses specified; this, however, is not always necessary (e.g., see broadcast channels below).

- **Type** The channel type determines degree of connectivity, resolution to physical interfaces, and data direction for channels.

  The degree of connectivity is one of *point-to-point, point-to-multipoint, broadcast.* The choice of point-to-point channels are obvious; point-to-multipoint channels provide a mapping to MAC layer multicast (e.g. on Ethernet interfaces). Broadcast channels provide physical layer broadcast and are useful for auto-configuration of network nodes and for initializing certain network protocols.

  As mentioned above, channels are logical entities and are not mandated to be bound to particular physical interfaces. However, certain channels (e.g. broadcast and point-to-multipoint channels) must resolve to physical interfaces at channel creation time — we term these *absolute* channels. *Relative* channels do not have an associated physical channel at channel creation time — the upper layer routing and protocol stack selects a physical interface when data is transmitted on a relative channel. The selected interface for a logical channel may change during the lifetime of the channel: a point-to-point relative UDP channel may choose different interfaces at different times for the same address depending upon the state of IP routing tables.

  The channel direction is one of *incoming, outgoing*, or *bi-directional.*

- **Attributes** Associated with each channel is a named set of attributes. No specific attribute is necessary, and the set of attributes can change over the lifetime of the channel. For the most part, attributes are used by higher layers to select channels for data input. As an example, the `{GZip|DES|UDP|IP|Ethernet}` channel could have an attribute "Secure" to indicate that it may be selected for secure data transmission.

- **Filters** Packet filters provide a mechanism for classifying incoming packets. Bowman channels use user-specified filters to segregate and demultiplex each incoming packet. The filters are dynamic (i.e. the set of filters and filter expressions may be changed during the operation of the channel); Section Section 6.2 [Packet Classifier], page 31 describes further the operation of the packet filters.

- **Status** Bowman channels may exist without participating in communication. The channel's status determines whether it may be used to transfer data. Users can supply filters

[3] This assumes that there exists a protocol module `GZip` that performs data (de-)compression and a protocol module `DES` that performs data encryption and decryption.

to channels that have been created but whose status is set to *down*; matching packets will be available when the channel's status is set to *up*.

- **Queues** Bowman channels support per-user, per-channel output queues. Per-channel queues are useful because channels eventually map to physical interfaces and per-channel queues make bandwidth allocation on physical links easier. Though not implemented, we envision a hierarchical scheduling policy in which each channel using a link is given a certain share of the link bandwidth; bandwidth allocated to the channel is then distributed to its users using the channel scheduling policy. An open research question concerns relative (unbound) channels: it is not clear how to efficiently guarantee bandwidth on the outgoing physical link to channels that do not resolve to the same outgoing interface for each packet.

  Bowman specifies per-user queues for the efficient support of application-specific congestion-control and other active applications in which data for a particular user may have to be specially processed or evicted from a congested output queue.

  The channel schedule policy for queues is not specified, however the architecture document *ArchDraft* does suggest work-conserving queuing policies in order to conserve node resources and maintain network stability.

- **QoS** The quality of service parameter for Bowman channels is intended to provide per-channel per-user guarantees using queue scheduling policies. Currently, however, it is neither specified nor implemented.
- **Access** The access parameter for the channel provides a mechanism for implementing security policy on a per-channel basis. The current Bowman implementation does not identify security principals in the system and does not implement access restrictions.

3.2.2 Channel API

3.2.2.1 Channel Information

The structure `o_channel_info` captures all the fields described in the previous section. This structure is used to define a channel during creation.

```
typedef struct o_channel_info_t
{
  o_channel_type         type;        /* Type (Connectivity, etc.) */
  o_channel_dir          dir;         /* Direction of flow of packets */
  o_channel_status       status;      /* Alive (usable) or not */
  o_channel_access       access;      /* Privileges */
  o_channel_addr_list    addr_list;   /* Set of remote addresses */
  o_channel_attr_list    attr_list;   /* Set of attributes */
  o_channel_pattern      pattern;     /* Identify packets on this channel */
  o_channel_proto_stack  proto;       /* Protocol stack */
  o_proto_param          param;       /* Parameters for Protocol processing */
} o_channel_info;
```

Each of the fields in `o_channel_info` are described by corresponding structures as follows:

- o_channel_type: The generic "type" byte of a channel includes:

•connectivity •plumbing •OS support

```
typedef u_char o_channel_type;
```

Allocation of the bits in the "type" byte and specific macros are described in the header files "o_channel.h"

• o_channel_dir: Direction of the flow of packets through the channel

•Bi-direction: either way •Uni-incoming-direction: this channel receives packets. •Uni-outgoing-direction: sends packets out on this channel.

```
typedef enum {
  o_uni_incoming_dir,
  o_uni_outgoing_dir,
  o_bi_dir
} o_channel_dir;
```

• o_channel_status: Status to indicate whether the channel is alive or not. Even though a channel is created, it cannot be used unless the status flag is set.

```
typedef enum {
  o_up,
  o_down
} o_channel_status;
```

• o_channel_access: Access privileges to ensure security in modifying the characterisics of a channel.

```
typedef enum {
  o_privileged,
  o_regular
} o_channel_access;
```

• o_channel_addr_list: A channel can be associated with multiple remote end points and all their addresses are captured by this address list. But in this release, this field has no significance as we store the addresses of remote end points in protocol parameters fields.

```
typedef struct o_channel_addr_list_t
{
  generic_addr ga_list[_O_CH_MAX_ADDRS];
  int          num_addrs;
} o_channel_addr_list;
```

• o_channel_attr_list: Aflows access a channel by specifying an attribute. This list of attributes is used for channel selection.

```
typedef struct o_channel_attr_list_t
{
  char  name_attr[_O_CH_MAX_ATTRS][_O_CH_MAX_NAME];
  int   num_attrs;
} o_channel_attr_list;
```

• o_channel_pattern: A description of the link headers that will be used to divert packets to this channel. This pattern will be put in the demux right before the channel protocol processing.

```
typedef struct o_channel_pattern_t
{
```

```
    char desc_filter[_O_CH_MAX_STRING_LEN];
  } o_channel_pattern;
```

- o_channel_proto_stack: The set of protocols that form a combination (stack) to do the processing of packets on this channel.

```
  typedef struct o_channel_proto_stack_t
  {
    char proto_name[_O_CH_MAX_NAME];
  } o_channel_proto_stack;
```

- o_proto_param: This structure contains the protocol specific parameters used during the protocol processing especially during the encapsulation/decapsulation of the protocol headers. This structure is in the header file "<sys/o_proto.h>".

Every channel in an Odyssey node has a corresponding entry in the channel table that has the following type:

```
typedef struct o_channel_t
{
  u_int               id;             /* Descriptor of the channel */
  o_channel_info      info;           /* Characteristics of channel */
  o_pf_handle         pfh;            /* Demultiplexing Handle */
  o_protocol         *pspec;          /* Protocol Functions */
} o_channel;
```

3.2.2.2 Channel Creation

The call `o_channel_create` takes the description of the structure of a channel in the argument `info` and returns a descriptor corresponding to the new channel that is created.

```
u_int o_channel_create(o_channel_info *info)
```

3.2.2.3 Channel Modification

The characteristics corresponding to the structure of a channel can be retrieved or modified using the `o_channel_get_xxx` and the `o_channel_set_xxx` calls. Attributes (in the form of character strings) are added to channels using the `o_channel_add_attr` call.

```
boolean             o_channel_get_info(u_int c_id ,
                                       o_channel_info *info);

o_channel_type      o_channel_get_type(u_int c_id );
boolean             o_channel_set_type(u_int c_id ,
                                       o_channel_type type);

o_channel_status    o_channel_get_status(u_int c_id );
boolean             o_channel_set_status(u_int c_id ,
                                         o_channel_status status);

o_channel_dir       o_channel_get_dir(u_int c_id );
boolean             o_channel_set_dir(u_int c_id ,
                                      o_channel_dir dir);
```

```
o_channel_access    o_channel_get_access(u_int c_id );
boolean             o_channel_set_access(u_int c_id ,
                                         o_channel_access access);

o_channel_attr_list o_channel_get_attr_list(u_int c_id);
boolean             o_channel_add_attr(u_int c_id, char *oca);
```

3.2.2.4 Channel Selection

An aflow can select a channel for data input, by specifying a set of attributes using the function call `o_channel_get_match` This call takes in a set of attributes `exp` as an argument and returns a list of all channels `match_list, num` that match the requirement specified by the aflow.

```
int o_channel_get_match (u_int *match_list, u_int *num)
```

3.2.2.5 Channel Subscription

After an aflow selects a particular channel, it can specify a filter for the packets that it expects as input data. This filter is used for demultiplexing arriving packets to the appropriate aflows. The function call `o_channel_subscribe` takes a channel descriptor `c_id`, a filter expression `filter` and an aflow descriptor `demux_id` as arguments.

```
o_channel_subscribe(u_int c_id, char *filter, u_int demux_id)
```

3.3 Aflows

In Odyssey, any per flow processing takes place in an "aflow". These are analogous to processes in Unix.

o_aflow_id: A unique identifier to represent an aflow in the AFlow Space, especially to access an aflow specific values in the global aflow table. Also used in logging messages written out by an aflow.

```
typedef u_int  o_aflow_id;
```

o_aflow_thread_id: Identifies the light weight process thread inside an AFlow.

```
typedef u_int  o_aflow_thread_id;
```

o_aflow_CBE: Represents an entry point in the code to run an AFlow. The function prototype takes only one argument (void *) and returns void. This code block entry point is called when a separate pthread is created for this AFlow.

```
typedef void  (*o_aflow_CBE)(void *);
```

CBE : Code Block Entry o_aflow_thread_type: Two kinds of threads, ones that are joinable and others which are detached. Maps exactly into pthreads types.

```
typedef enum o_aflow_thread_type{
  o_aflow_joinable,
  o_aflow_detached
}o_aflow_thread_type;
```

o_aflow_arg_type: Since we have only one argument possible for the code block entry function of the AFlow, we have this structure, that can accomodate variable arguments.

```
typedef struct o_aflow_arg_type
{
  int    argc;
  char   argv[_O_AFLOW_MAX_ARGS][_O_AFLOW_ARG_SIZE];
} o_aflow_arg_type;
```

Method interface to the Aflows

Initialises the global aflow table and also initialises the thread space

```
void        o_aflow_init();
```

To create an aflow with the name, code block entry function, an argument to that function, size of the input queue, any credentials

```
o_aflow_id  o_aflow_create(char *name,
   o_aflow_CBE fn,
   o_aflow_arg_type *arg,
   u_int InQ_size,
   void *cred);
```

A default version of the above without the input queue size and credentials of the aflow

```
#define     o_aflow_create_default(n,f,a) o_aflow_create (n, f, a, 0x0, 0x0)
```

To execute an aflow by referencing an id from the global aflow table

```
boolean     o_aflow_run(o_aflow_id id);
boolean     o_aflow_destroy(o_aflow_id id);
```

Each Aflow has an input queue as part of its data structures and it can read out the first packet from the queue by calling "o_aflow_read". It can flush all the packets by invoking "o_aflow_flush".

```
o_pkt      *o_aflow_read ();
void        o_aflow_flush();
```

To obtain the self identification of the Aflow and the thread number inside that Aflow

```
o_aflow_id  o_aflow_getid();
o_aflow_thread_id  o_aflow_thread_getid();
```

A very useful logging facility, that writes a message based on the boolean flag (given as first argument) on to a log book (right now, it is the "stdout"). It is Multithreaded Safe than other variants of printf. This function resolves concurreny in logging by using a mutex.

```
void        o_aflow_log (boolean c, ...);
```

To set the name of an aflow

```
boolean     o_aflow_set_name(char *name);
```

Given an expression, to get a list of aflows whose names match the expression. Also return the number of matches too

```
int         o_aflow_get_match(char *exp, u_int *match_list,
      u_int *num);
```

To the thread pool associated with an Aflow, add another thread, with the code block entry function, an argument to that function and the type of the thread

```
int         o_aflow_add_thread(o_aflow_CBE f,
       o_aflow_arg_type *arg,
       o_aflow_thread_type tt);
```

Run a thread from the thread pool of this aflow

```
boolean     o_aflow_run_thread(u_int tid);
```

To exit gracefully, after reclaiming the resources used by an aflow.

```
void        o_aflow_exit ();
int        _o_aflow_kill_others();
```

3.4 State-store

The Odyssey state store provides a tuple space for per-flow data.

Odyssey stores are of type `o_tuple_space`. Stores are allocated with the `o_ss_allocate` call.

```
o_tuple_space  *o_ss_allocate();
```

There is an analogous `o_ss_deallocate` call to deallocate a store. The tuple space `ts` can no longer be used. Further, all associated tuples in the store are lost. (Of course, if a pointer was inserted as a value in the store, it is still valid, as long as there is a mechanism to refer to the value of the pointer).

```
void            o_ss_deallocate(o_tuple_space * ts);
```

The basic mutator of a state store is the `o_ss_put` call. The arguments are a pointer to a tuple space, a key (which is a string of characters) and an opaque value. The precise definitions of the key and value parameters are:

```
typedef char          *o_tuple_key;
typedef void          *o_tuple_val;
```

Note well: Four bytes *pointed to* by the value `v` is put in the state store `ts`.

```
void            o_ss_put(o_tuple_space *ts, o_tuple_key k, o_tuple_val* v);
```

Values put in a store are retreived via the `o_ss_get` call. The arguemnts are a store identifier and a key. This call will return NULL if no value associated with the key `k` is in the store `ts`.

```
o_tuple_val    *o_ss_get(o_tuple_space *ts, o_tuple_key k);
```

The `o_ss_remove` call can be used to remove the value associated with a key from a store. This is equivalent to associating a NULl value with the given key.

```
void            o_ss_remove(o_tuple_space *ts, o_tuple_key k);
```

Bowman provides native support for named registries. Registries provide a mehcanism for compuations to share data without sharing specific variables.

The basic operation of registries is as follows: One flow creates a named registry (say "Burdell-registry"). Then items can be added or retrieved to/from the registry by referring to it just by the name "Burdell-registry". Thus a different flow could put items in the regisrty just by referreing to the registry by name.

Further, items in registries are given unique indices — this is useful as a mechanism to get total ordering on a set of datum that is held by different flows.

The registry implementation is MT-safe. If two different flows create a registry of a given name, one and only one is guaranteed to succeed. Further, indices provided to items in a registry are guaranteed to be unique and mototonic.

Registries are created using the `o_ss_new_registry` call. This call guarantess that only one regisry of a given name can be created. The input is the name for the registry. The return value is always TRUE (as either the registry is created or a registry by that name already exists. The `o_registry_exists` call can be used to query whether a registry of a given name exits already.

```
boolean         o_ss_new_registry(char *r);
int o_ss_registry_exists (char *r);
```

The `o_ss_incorporate_in_registry` call is the basic mutator for a registry. The inputs are a registry name, a key, and value.

If the named registry does not exist, this call will fail and return 0. Else, the item is incorporated into the registry and an unique index of the item in the registry is returned. The index is guaranteed to at least 1.

```
int             o_ss_incorporate_in_registry (char *r, char *it, void *val);
```

The `o_ss_lookup_in_registry` and `o_ss_in_registry` calls retreive information from the registry. The inputs to each are a registry name and an item key.

Both calls will fail (and return 0) if the registry name is invalid.

The `o_ss_lookup_in_registry` call returns the *value* associated with the item from the named registry. It will return NULL if such an item did not exist in the registry.

The `o_ss_in_registry` call returns the *index* of the item from the named registry. It will return 0 if such an item did not exist in the registry.

```
void           *o_ss_lookup_in_registry (char *r, char *it);
int             o_ss_in_registry (char *r, char *it);
```

The `o_ss_put_in_registry` call is used to change the value of an item already in a given registry. It will return 0 if the registry name is invalid.

Otherwise the index of the item is returned and the value is set to the input parameter `v`. Note that if the item did not exist in the registry, the put call will call the incorporate call and return the new index.

```
int             o_ss_put_in_registry(char *r, char *it, void *v);
```

3.5 Bowman Extension Mechanism

See Section 4.1 [Extensions Overview], page 22

3.6 Odyssey Packet API

Odyssey packets are used to send and recieve on channels. Packets have no meaning between Odyssey nodes or outside of an Odyssey node. They are only a data structure for use within the node, they do not specify a wire format. Bowman and CANEs use the same packet definition. The Odyssey packet contains a from field which is the channel id which the packet was first received at the node. The buffer contains the contiguous string of bits carried by the channel. The data_off field specifies the first valid byte in the buffer. Data in the buffer will be word aligned at the data offset. size bytes from data_off are valid. If a packet is created, data_off should be initialized to at least O_PKT_DEFAULT_HDR_SIZE, to allow for output channel processing.

```
typedef struct o_pkt_t
{
  char           buffer[O_PKT_MAX_PKT_SIZE - 56]; // aligned
  /* The following four fields are used in packet memory management in
     Odyssey. "big buffer" denotes a pool of packets that are
     pre-allocated and managed to hold packets of different flows */

  int            over_flow_pattern;

  int            myIndex;        /* Index of the current elt in big buffer */
  int            prevFreeIndex;  /* prev free elt in the big buffer */
  int            nextFreeIndex;  /* next free elt in the big buffer */
  boolean        isFree;         /* to denote the occupancy of this elt */


  u_int          from; /* channel id */
  u_int  sub; /* sub-channel ID; for example filter ID */
  int            ref_count;
  u_int          data_off;   /* Offset into the buffer to the valid data */
  u_int          size;       /* Length of valid data */
  /* Moved the following time fields to the end of the packet for
     reasons of overflow, etc.. */
  struct timeval time;
  long long      t_be;  /* time stamp before enqueueing into the queue */

} o_pkt;
```

Returns a valid initialized packet, or NULL if no more packets are available.

```
o_pkt     *o_pkt_get();
```

Decrements the reference count, and deallocates the packet if the reference count is 0. Since o_pkt duplicate is disabled the reference count is largely unused.

```
boolean    o_pkt_free(o_pkt *op);   /* Free only if ref count is zero */
```

This returns a new packet with identical values to the one given, including the buffer.

```
o_pkt     *o_pkt_copy(o_pkt *op); /* allocates memory and copies */
```

Duplicate currently does the same thing as copy. In the future we hope to define a duplicate/reference count semantic which will facilitate some no-copy scenarios.

```
o_pkt     *o_pkt_duplicate(o_pkt *op); /* only increments reference count */
```

3.7 Generic Addresses

Odyssey uses a generic address structure that includes different types of address formats that are used in the protocol stack.

The generic address types in Odyssey are enumerated as follows:

```
typedef enum {
  addrtype_ether,
  addrtype_ip,
```

```
    addrtype_dns,
    addrtype_none,
  } generic_address_type;
```

The *generic_addr* structure itself is a union of different address types.

```
  typedef struct generic_addr_t {
    generic_address_type t;
    int l;
    union {
      struct ether_addr eth;
      struct in_addr    ip;
      struct dns_addr   dns;
    } addr;
  } generic_addr;
```

The generic address type is used in storing the local and remote addresses of channels, the information about local interfaces.

3.8 Threads API

The Odyssey Threads Interface.

Odyssey threads are a simplified interface to POSIX Threads (`pthreads`).

Thread creation is handled entirely through the aflow interface.

The only thread specific interfaces visible outside Bowman's core has to do with synchronization and per-flow private data.

The main primitives for synchronization are mutexes and condition variables.

They are declared as follows:

```
     typedef pthread_mutex_t o_thread_mutex;
     typedef pthread_cond_t o_thread_cond_var;

     typedef pthread_key_t o_thread_private_data;

  #define o_thr_mutex_init(m,a) pthread_mutex_init(m,a)
  #define o_thr_mutex_destroy(m) pthread_mutex_init(m)
  #define o_thr_mutex_lock(m) pthread_mutex_lock(m)
  #define o_thr_mutex_trylock(m) pthread_mutex_trylock(m)
  #define o_thr_mutex_unlock(m) pthread_mutex_unlock(m)

  #define o_thr_condvar_init(c,a) pthread_cond_init(c,a)
  #define o_thr_condvar_destroy(c) pthread_cond_destroy(c)
  #define o_thr_condvar_broadcast(c) pthread_cond_broadcast(c)
  #define o_thr_condvar_signal(c) pthread_cond_signal(c)
  #define o_thr_condvar_wait(c,m) pthread_cond_wait(c,m)
```

The Private data primitives provides a mechanism for thread-specific variables. Note that a private data variable is of type `o_thread_private_data` and should be allocated only once. Once again these calls are wrappers around their pthread counterparts.

```
int o_thr_allocate_private_data (o_thread_private_data *pd);
int o_thr_private_data_set (o_thread_private_data *pd, void *v);
void *o_thr_private_data_get (o_thread_private_data *pd);
```

3.9 Random Numbers

The Bowman random number generators are aliases for the POSIX `srand` and `drand48` functions.

```
void   o_seed (long seed);
double o_rand ();
```

3.10 Odyssey Time

The Bowman time functions define the `o_timeval` structure. It is defined as

```
typedef struct timeval o_timeval;
```

The two functions provided are `o_gettimeofday` and `o_time_string`. `o_gettimeofday` is aliased to the POSIX `gettimeofday` function with a NULL timezone value.

`o_time_string` formats the input character string with the current time `hh:mm:ss`.

```
int  o_gettimeofday (o_timeval *tv);
void o_time_string (char *s);
```

3.11 Registry for text interfaces

The Info registry contains callbacks for addition to and lookup into the info registry, as well as a structure which defines the contents of a registered item. Modules such as channels, OIF, and other extensions register an info structure in order to export information to a text interface. Text interfaces such as HAL use the registry query commands and call functions specific to those commands.

This structure contains the name of the command that should be associated with the structure, the help message which should define the use of the list and show functions. The operation of the list and show functions are user define; however, the intention is for list to reveal an indexed table with minimal detail on each element of the table. The show command accepts an index, and is intended to reveal more detail than the list command did on the indexed entry.

```
typedef struct info_struct{
  char namestr[128];
  boolean (*list_func)(char *);
  boolean (*show_func)(char *, int);
  char helpmsg[1024];
} info_struct;
```

This function is used to register a completed info structure. Doing so reveals the structure and its functions to info users, like HAL.

```
boolean o_info_put(char *, info_struct *);
```

This function is used to find registered info structures.

```
info_struct *o_info_get(char *);
```

4 Extensions

4.1 Overview of Extensions

To extend the capability of the Bowman operating system in Odyssey, we have provided a generic mechanism to dynamically load a module. This *extension* interface allows new modules to be registered with the global extension table. The modules can either export a new set of function calls or can be used as an *aflow*. Some of the modules that are needed to extend the operating system interface are loaded during boot-up phase by reading a configuration file. (The configuration file is passed as a command-line argument to Bowman).

Each module is recorded in the "o_mod_desc" structure. It includes a name for the module, major and minor version numbers, a description of the module.

```
typedef struct o_mod_desc_t
{
  struct o_mod_desc_t *next, *prev;
  u_int        id;
  o_dl_desc    handle;
  char         name[32];
  u_char       maj_ver;
  u_char       min_ver;
  char         description[128];
}o_mod_desc;
```

Each extension should have a default function that is used as an entry point to get a handle to the extension. This function is named as "_entry" and take a single argument of type "u_int". Usually, the descriptor/handle of the loaded module is passed as the argument to the entry function.

```
#define o_ext_entry_point "_entry"
typedef void (*o_ext_entry_fn) (u_int);
```

Initialises the global extensions table.

```
boolean     o_ext_init();
```

To set the path, where to look for modules. This path is used to load the extensions listed in the *init* file to Bowman.

```
void        o_ext_set_path(char *path);
```

To dynamically load and unload a module. These modules are typically shared objects compiled separately from the regular Operating System. They can use the symbols exported by OS, but if they refer to symbols in separate libraries, they need to be resolved before they are loaded into Odyssey.

```
o_dl_desc   o_ext_load(char *lib_name);
o_dl_desc   o_ext_unload(u_int id);
```

To list all the loaded modules.

```
o_mod_desc *o_ext_list();
```

Each module should call this register function in its default entry function (with name "_entry"). This arguments to this function include the handle to this module, a name for this module (to be used later as key to access this module), major and minor version numbers, a brief description of the module. The return value of the function is an index into the global extensions table.

```
u_int       o_ext_register(o_dl_desc h, char *name, u_char major_ver,
     u_char minor_ver, char *description);
```

4.2 Route Table Extension

4.2.1 Data Structure of Routing Table

A routing entry consists of a routing key and a channel identifier to which the packet with the matching routing key should be directed.

```
typedef struct o_rt_entry_t {
  o_rt_key key;
  u_int ch_id;
}o_rt_entry;
```

The routing key is a <value, length> tuple. The value could be IPv4 address or DNS name of the destination. The routing key is interpreted in the context of a specific routing table type.

```
typedef struct o_rt_key_t {
  char val[MAX_VAL_LENGTH];
  u_int val_len;
}o_rt_key;
```

4.2.2 Interface to the routing table

A routing table is created with `o_rt_create()` call with `name` and `type` arguments. Type can be `ip`, `dns`, etc.

```
int o_rt_create (char *name, char *type)
```

To get an identifier of the routing table with its name, `o_rt_get_id()` call is used. Upon success, it returns the index of the routing table. Otherwise, it returns -1.

```
int o_rt_get_id (char *name)
```

`o_rt_get_rt()` call returns the pointer to the routing table with routing table index `rt_id`.

```
o_rt *o_rt_get_rt (u_int rt_id)
```

To destory a routing table, `o_rt_destroy()` call is used. It returns TRUE upon successful destroy.

```
boolean o_rt_destroy (u_int rt_id)
```

To add a routing entry to the routing table, `o_rt_add_entry()` is used. It adds the routing entry `entry_p` to the routing table with index `rt_id`.

```
void o_rt_add_entry (u_int rt_id, o_rt_entry *entry_p)
```

Deletion of a routing entry from a routing table is performed with `o_rt_del_entry()` call.

```
void o_rt_del_entry (u_int rt_id, o_rt_entry *entry_p)
```

Routing table lookup is done by `o_rt_lookup()` call. It has two arguments: Index of routing table and pointer of the packet to be routed. Upon success, it returns a channel identifier to which the packet should be destined. Otherwise, it returns -1.

```
u_int o_rt_lookup (u_int rt_id, o_pkt *op)
```

4.3 Odyssey Code Loading

The code loader, AKA code fetch, is made of of several pieces: the code fetch demux, the code fetch client and the code server.

The demux portion allows any method of fetching code to be added, including the code fetch client. This is done by writing the new code fetching function which will take a pointer to a o_code_desc_type structure. The caller will have filled in the url. The function must fetch the code and return a boolean indicating success or failure. If the fetch was successful, the "location" member of the structure must be filled in with the pathname to the code.

The new function must be registered by filling in an o_code_fetch_type structure with a unique name and the pointer to the function, then calling o_cf_register in the _entry function. See code_client.c for an example of adding a fetch function.

The code client portion is added as an extension to bowman by inserting the following line in the bowman init file: load "/admin/lib/solaris2.sparc/code_client.so"

When it is loaded, a file can be fetched by sending a url of the form: ocf://<server-name><pathname>

For example, ocf://foo.cc.gatech.edu/tmp/o_code/bar.so would retrieve the file /tmp/o_code/bar.so from server foo.cc.gatech.edu using the ocf code fetch method. The client checks to see if it has the code cached locally before checking with the See Section 6.4 [Code Server], page 34.

4.4 Interface to address resolution

OIF is the Odyssey interface interface, it contains structures for each physical interface on a node. The structure for each interface contains the addresses bound to an interface.

The o_if structure contains the interface name, the number of generic addresses bound to the interface, and an array of generic addresses.

```
typedef struct o_if
{
  struct o_if *next;
  struct o_if *prev;
  char if_name [255];
  u_int _addrs;
  struct generic_addr_t addr [_OIF_NUM_ADDRS];
} o_if;
```

This function will return a pointer to the list of interfaces.

```
struct o_if *oif_list_ifs ();
```

This function returns a pointer to the interface with the give name. If no interface matches the name it returns NULL.

```
struct o_if  *oif_get_if(char *if_name);
```

This function returns a pointer to the created interface with the give name.

```
struct o_if *oif_new_if (char *if_name);
```

The following functions can be used to add and remove generic addresses to an existing interface.

```
boolean oif_add_addr (struct o_if* oif, generic_addr *addr);
boolean oif_delete_addr (struct o_if* oif, generic_addr *addr);
```

Use this to find an interface associated with the generic address given.

```
struct o_if  *oif_find_if (generic_addr *addr);
```

Use this to get an address of type addrtype associated with the physical interface that ch_id is bound.

```
boolean oif_find_addr_by_channel (int ch_id, int addrtype,
  generic_addr *ga);
```

OIF currently learns the interfaces from a flat file located in $ODYSSEY_ROOT/bowman/etc/oif.<host>, where <host> is the name of the host where bowman is running. In the future we will dynamically query the OS for this information.

4.5 Heavy Weight Timers

The *HWT* — Heavy Weight Timer module provides threaded timers. Each aflow has a special timer thread that executes timer functions.

Each aflow initializes its timer thread by calling `hwt_init`. This should be called only once.

```
void          hwt_init (void);
```

The `hwt_start_timer` function is used to start a timer with a specified frequency (ticks), with the function to be called upon a time out and the argument to that timeout function as the other arguments

The `hwt_start_timer` function returns an opaque timer handle.

```
hwt_handle_t  hwt_start_timer(int ticks, void (*func)(), int arg);
```

Using the opaque timer handle, an already established timer can be canceled by calling the `hwt_cancel_timer` function.

```
boolean       hwt_cancel_timer(hwt_handle_t);
```

The `hwt_fn_init` function initializes the heavy weight timers for an aflow (much like the `hwt_init` function). The `_hwt_fn_init` function, however, also allows the caller to specify a function `f` of type `hwt_init_fn` and an argument `a`. This specified function `f` is called with the given argument *once* before any timeout functions are executed. All timer thread specific initializations should be done within the specified function (`f`).

```
typedef void(*hwt_init_fn)(int);
typedef struct _hwt_init_fn_arg
{
  hwt_init_fn f;
  int a;
} _hwt_init_fn_arg;
void    _hwt_fn_init (hwt_init_fn f, int arg);
```

5 Node Administration

5.1 Abstract Link Manager

Abstract Link Protocol (ALP) provides a mechanism to create or destroy abstract links. A Bowman initialization script starts Abstract Link Manager (ALM) as an aflow that processes ALP messages and manages abstract links on each node using the channel interface. The UDP port number for the ALM is 9998. Sending ALP messages to the ALM causes channels to be created at Bowman nodes.

The following is the syntax of abstract link description for ALP messages. An ALP message can contain multiple abstract link descriptions.

```
([create|destroy] {
                    (name LinkName)
                    (   proto [ether | udp]
                        (host (addr HostName) ([dev-name Device Name |
                                                port PortNumber])
                        )
                        (host (addr HostName) ([dev-name Device Name |
                                                port PortNumber])
                        )
                    )
                    (status [up | down])
                    (dir [bi | uni-incoming | uni-outgoing])
                    (type [point-to-point | multi-point | broadcast |
                           absolute | relative | raw])
                    (access [regular | privileged])
                  }
)
```

The following is an example ALP message. This ALP message creates a full-duplex point-to-point UDP link named "AE-35" between the machines with DNS names *Discovery.Odyssey.Net* and *Earth.Odyssey.Net* and a full-duplex point-to-point Ethernet link named "AE-36". Both the name "AE-35" and the attribute "Unreliable" will be added as attributes to the resulting channels for the UDP link at both Bowman nodes Discovery and Earth.

```
(create   {
              (name AE-35)
              (
                   proto udp
                   (host (addr Discovery.Odyssey.Net) (port 9000))
                   (host (addr Earth.Odyssey.Net) (port 9000))
              )
              (status down)
              (dir bi)
              (type point-to-point)
              (type relative)
              (access regular)
              (attribute Unreliable)
```

```
          }
)
(create   {
             (name AE-36)
             (
                  proto ether
                  (host (addr Discovery.Odyssey.Net) (dev-name hme0))
                  (host (addr Earth.Odyssey.Net) (dev-name hme0))
             )
             (status down)
             (dir bi)
             (type point-to-point)
             (type raw)
             (access regular)
             (attribute Unreliable)
          }
)
```

5.2 The ARP service

The ARP service provides access to mac addresses given a dns name. It does not provied RARP functionality at this point. It does require root access as it uses an ether channel to send and receive arp requests and responses. The ARP service stores all entries without a time limit, but it does update mac addresses as the are published on the ethernet.

This is an arp table entry, it contains the mac address, a pending flag which is set when a query is waiting for a response. The interface which the response was received on, and a condition variable which can be waited on if the pending flag is set. The condition variable and pending flag should ONLY be used by arpQuery.

```
typedef struct __arp_table_entry
{
  u_char macAddr[6];
  boolean pending;
  struct o_if *interface;
  o_thread_cond_var condVar;
} _arp_table_entry;
```

arp Query will return true if the dnsAddress resolves to an ethernet address. If the dns address is not in the arp table, the arp writer thread will be signaled, and an arp request will be generated. If the response is not recieved in three seconds by the arp writer then the query will return false. If the response is received, the writer will signal the query and the address will be read from the table and returned to in ethAddr.

```
boolean arpQuery(char *dnsAddr, u_char *eth_addr);
#define o_dns2eth(dnsAddr, ethAddr) arpQuery(dnsAddr, ethAddr)
```

This returns a COPY of the arp table entry for a dns address.

```
boolean arpGetEntry(char *dnsAddr, _arp_table_entry *ARPEntry);
```

5.3 Abstract Topology manager

Abstract Topology Protocol (ATP) provdies a mechanism to create or destroy abstract topology. A Bowman initialization script starts Abstract Topology Manager (ATM) as an aflow that processes ATP messages and manages abstract topologies on each node. The UDP port number for the ALM is 9999. Sending ATP messages to the ATM causes abstract topologies to be created at Bowman nodes. Bowman also supports multiple abstract topologies simultaneously.

The following is the syntax of abstract topology description for ATP messages.

```
(name TopolgyName)
([add|delete]  LinkName)
([add|delete]  LinkName)
...
```

The following is an example ATP message. This ATP message creates an abstract topology named "Bowman" that consists of two abstract links, "AE-35" and "AE-36" between the machines with DNS names *Discovery.Odyssey.Net* and *Earth.Odyssey.Net*.

```
(name Bowman)
(add    AE-35)
(add    AE-36)
```

5.4 Routing Table manager

Bowman provides a static link-state unicast routing protocol. A Bowman initialization script starts a routing protocol as an aflow that processes routing control messages and creates routing tables on each node. The UDP port number for routing protocol manager is 9990. The routing protocol runs Dijkstra's shortest path algorithm over a specified abstract topology and generates a routing table at each Bowman node that participate in the abstract topology. Since Bowman supports multiple abstract topologies, there are multiple routing tables that correspond to the multiple abstract topologies. The routing protocol also supports multi-homed hosts that have multiple physical interfaces. However, the information for the multi-homed hosts should be provided to the routing protocol since the routing protocol is static and does not know the existence of such multi-homed hosts.

The routing protocol provides the following commands.

- **(multi_home Host1 Host2 [Host3 ..])**
- **(calculate AbstractTopologyName RoutingTableName)**

The following is an example routing control message. This message creates a routing table "Bowman_rt" for the abstract topology "Bowman". The "multi-home" is a command that tells the routing protocol that the following list of hosts are in fact multi-homed single host. There could be multiple list of multi-homed hosts and the list should be placed before "calculate" command that triggers the routing protocol to run routing algorithm.

```
(multi_home
        Discovery.Odyssey.Net
        Discovery-1.Odyssey.Net
        Discovery-2.Odyssey.Net
)
(calculate Bowman Bowman_rt)
```

5.5 CANEs Signaling and User Interface

The CANEs signaling interface implements the CANEs User Interface (*CUI*) signaling protocol. CUI messages are sent on a predefined port and consist of a CUI message specified in the CUI language. The CUI message is in two parts: computation and communication.

The computation part defines a directed acyclic graph (DAG). The root of the DAG is an underlying program. Each child node corresponds to an injected program bound to a slot in its parent. The arcs in the DAG identify the slot to which the child is bound. Each node contains enough information for the EE to fetch the code required to execute the computation DAG.

The communication part of the CUI message identifies a routing schema and a Odyssey topology for the user. Further, the user can specify a set of packet classifiers that is used to select incoming packets that the flow acts on. This communication part provides enough information for the CANEs EE to establish the input and output channels required for the user's computation.

We provide a definition for the CUI language and an example of its use next.

```
MESSAGE : ( flow <flow-name> COMPUTATION COMMUNICATION )+

COMPUTATION : (DATA_NODE)+ ( PROGRAM_NODE )+
```

A CUI message consists of a flow name, specifications for a computation, initialization data and aflow input and output for communication. The specification of the computation defines a DAG with the underlying program as the root.

```
DATA_NODE : data DATA_TUPLE

DATA_TUPLE : (<data_key> <data_value>)
```

Each node (`DATA_NODE`) defines a (`DATA_TUPLE`), which is a key, value pair. This is an argument passing mechanism for injected programs. The `data_value` can be retrieved by ALL injected programs in the flow by using the `canes_get` call, with an argument of `data_key`.

```
PROGRAM_NODE : prog <prog-name> loc <location> code <entry-function> SLOT_LIST

SLOT_LIST : ( bind <slot-name> ( PROGRAM_NODE )+ )+
```

Each node (`PROGRAM_NODE`) is defined by the name of the program (specified after keyword `prog`), a location to fetch the program from (location specified after the keyword `loc`) and a function that is invoked to start the computation (specified after keyword `code`). Each node also specifies a list of slots to which other programs are bound. These slots must be defined by the program specified in the node. The slot list specifies the slot name and the root of another sub-DAG. Note that multiple programs can be bound to the same slot.

```
COMMUNICATION : ( io CHANNEL FILTER_LIST ROUTING_TABLE )

CHANNEL : ( channel <channel-name> )

FILTER_LIST : ( filter <filter-string> )+

ROUTING_TABLE : ( routing-table <rtab-name> )
```

The input specification contains a set of of channel names and and a list of filters for each channel. Packets that arrive on each of the specified channels and match *any* of the filters specified for that channel will be routed to the computation. The output specification is the name of a routing schema.

The `ROUTING_TABLE` variable is a legacy item & is ignored. Routing table names should be passed to route injected programs with the `DATA_NODE` argument passing mechanism described above.

5.6 Command Line Interface to Bowman

Hal is the command line interface to a node running bowman. A standard telnet to port 2001 on a bowman node will reveal the hal command prompt. Multiple users can connect to Hal at the same time. Hal uses the info registry to provide access to internal data structures. Currently there are only two modules which export functionality to the info structure, channel, and oif.

Hal recognizes the following commands:

```
help [<command>]
list <command>
show <command> <index>
quit
```

`<command>` is the string used to register the info structure, for example: channel, or oif. the list and show commands are bound to whatever pointer is given for list and show in the info structure. `<index>` is passed as an argument to the show function. Its meaning is defined by each function; however, in general the list command will show a table, and the show command will give detail on an indexed value in the table. The help message for a command is whatever is given in the helpmsg in the info structure. There is no way to query the available commands at this time.

**Currently Hal provides no user authentication.

6 Various notes on the distribution

6.1 Variable scoping within CANEs

6.1.1 Variable Sharing

The CANEs API provides a variable sharing mechanism that is used by injected programs to communicate with underlying programs. We adopt the following terminology: a "shared" variable refers to a variable that can be accessed by multiple programs. Underlying programs declare shared variables and "export" them (make them eligible for sharing). Injected programs declare "references" to shared variables (the space for the actual data for references must be declared by the underlying program). Further, injected programs must "import" shared variables exported by underlying programs (this finally creates the binding between the shared and the references variables). Further references, internally, are pointers to shared variables. Thus their values change synchronously each time the value of the shared variable changes in the underlying program, i.e. there is no need to export a shared variable each time its value changes.

Aside: CANEs injected programs can also export and raise slots to which "sub-injected" programs can be bound and so on. In effect, the slot exporting program (which may itself be an injected program) forms the *underlying* program to which injected programs can be bound. Note that variables are shared only between programs that export a slot and the set of programs immediately immediately bound to the exported slot. Variable sharing is constrained to one level of injected programs. Consider an example: Suppose U is an underlying program, I is a program bound to a slot in U, and J is a program bound to a slot exported by I. The program I can import variables exported by U. The program J can *only* import variables exported by the program I. However, it is possible for the program I to import a variable from U and subsequently export it. This variable, originally exported by U, and re-exported by I can now be imported by the program J. (End Aside).

There are two different types of sharing supported by CANEs. Multiple copies of an underlying program can be instantiated by different users concurrently. Each user-flow maintains a private copy of a *per-flow* shared variable. In this case, the sharing is between variable injected and underlying programs within the context of an user's flow. Shared variables can also be declared by underlying programs without the per-flow attribute. In this case, all the instantiations of the underlying program (and injected programs that bind to these instantiations) share a single copy of the variable.

6.2 Packet Classifier

6.2.1 Overview of Packet Classfier

Packet classification is an essential part of any flow-specific processing. In Bowman, the packet classifier is used to demultiplex packets to respective flows. Each flow specifies a set of filter rules that are applied on the incoming packets. Upon a succesful match, the packet is demultiplexed accordingly.

A filter rule is a set of `(offset, length, value)` tuples, that are used to match with the incoming packet buffer. The packet classifier is implemented as multi-way tree. Each node in the tree corresponds to a single tuple of a filter rule. A packet matching algorithm traverses the tree, starting from the root towards the leaf. At each intermediate node, it checks to see if the contents at `offset` of size `length` match `value`. Based on success or failure of this test, it branches to the next level of nodes in the tree. A complete match with a filter rule corresponds to a path from the root to the leaf of the tree. A demuliplexing key stored in the leaf node, identifies the flow that has installed the particular filter rule in the packet classifier.

6.2.2 Packet Classification

There are two little languages employed by the packet classifier, one to *specify* filters and one to *learn* about protocols.

Protocols are specified using a grammar with the `protocol`, `field`, etc. as keywords.

For example the following shows the C structure definition for Ethernet and how that is translated to something the classifier understands.

```
;struct ether_header {

; struct ether_addr ether_dhost;
; struct ether_addr ether_shost;
; u_short ether_type;
;}
(protocol ethernet
    (field proto offset 12 length 2)
    (field src offset 6 length 6)
    (field dst offset 0 length 6)
    (hdr_len 14)
)

(define ETHERTYPE_PUP 0x0200) ; /* PUP protocol */
(define ETHERTYPE_IP 0x0800) ; /* IP protocol */
(define ethertype_ip 0x0800) ; /* ip protocol */
(define ETHERTYPE_ARP 0x0806) ; /* Addr. resolution protocol */
(define ETHERTYPE_REVARP 0x8035) ; /* Reverse ARP */
(define ETHERTYPE_TRAIL 0x1000) ; /* Trailer packet */
(define ETHERTYPE_NTRAILER 16)     ;
```

Now, a filter can be specified using the tokens: `ethernet`, `proto`, etc. For example,

```
[ ethernet (proto = ETHERTYPE_IP) | *]
```

will match all ethernet segments that contain IP packets.

The `protocol-info` file in the pf/data directory contains definitions for Ethernet, IP, UDP, TCP, etc.

Note that each filter must be enclosed in square brackets and the last part of the filter must be a `| *`. The filter expression language is described in pf/packet-filter.[l|y].

In general, a filter expression contains a set of protocol names and predicates containing field names separated by a "|". Each time a "|" is encountered, the last defined header type is skipped.

Another example,

```
[ ethernet (proto = ETHERTYPE_IP) | ip (proto = IPPROTO_UDP) |
   udp (sport = 124) | *]
```

This filter will try to match for IP packets on the ethernet segments. If that succeeds, it will skip over the ethernet header and try to match on the proto field of the IP header. If the proto field is IPPROTO_UDP, it will then skip over the ip header, and match on the source port of the UDP header, etc. Note that this version does NOT implement variable header lengths, as such we cannot process IP headers with options.

The current configuration of the classifier returns the matching filter for which the maximum number of predicates match the given buffer.

For example, If a classifier had two rules, *viz.*

```
0: [ ethernet (proto = ETHERTYPE_IP) | *]
1: [ ethernet (proto = ETHERTYPE_IP) | ip (proto = IPPROTO_UDP) |
   udp (sport = 124) | *]
```

and an UDP packet with source port 124 were passed as the buffer, then the second rule would be matched. However, all other IP packets would match the first rule.

6.2.3 Packet Classifier API

A synopsis of the exported calls is provided below (out of the `share/pf/include/pf.h` file).

```
Filter *init_filter();
```

Returns a new packet classifier (with no rules defined).

```
int add_filter (Filter *ft, char *f, int demux_id);
```

Adds a new rule to a packet classifier (ft). The filter is specified in the packet filter language described below. When a packet matches this rule, the match_filter call will return the integer (demux_id). The demux_id should be non-negative. The return value for this call is an integer identifier for this particular filter rule.

```
int delete_filter(int fid, Filter *ft, int demux_id);
```

Deletes a filter rule (fid) in a classifier (ft). As a sanity check, the input integer demux_id must match the demux_id that was given when this rule was added to the packet classifier.

```
int match_filter (Filter *ft, char *buf);
```

Returns a demux_id if a char buffer starting at buf matches any of the filters specified in the classifier ft. Upon a match, the matching demux_id is returned. Otherwise, a defined constant NO_MATCH_INDEX (\equiv -1) is returned.

```
int add_pf_protocol (char *pinfo);
```

This call is used to "teach" a new protocol to the packet classifier.

6.3 Notes on Shared Utilities

Odyssey uses some common miscellaneous functions to perform useful tasks. These are not part of the Operating System, but nevertheless, very useful code that is shared across different modules of Odyssey. They are listed as shared utilities in Odyssey. Here are some of them with their descriptions:

- general : to allocate/deallocate memory
- genutils : for logging messages/warnings, file read, graceful exit
- netutils : for basic network programming tools
- ioutils : for input/output operations
- opt_parse : for parsing command line arguments/options
- queues : to implement singly/doubly linked circular queues
- selector : to manage the file descriptors used in asynchronous input/output using system call "select"

6.4 Notes on the Odyssey code server

The code_server is a completely separate executable which must be run on the machine which will send code to those nodes who request it. It listens to a well- known port to which the code_client connects. It accepts code requests and returns a header that either indicates (1) that code is on the way and the size of the code, or (2) an error message. If the code request can be satisfied, it will send the header, then the code, then an MD5 hash of the code. The code_client computes the hash on the code as it is being received, compares the hash to the one received and removes the code if there is a mismatch.

Currently the code client/server provides only a teency amount of security in that the server must speak the same protocol as the client. In the future, the md5 hash will be sent encrypted with a public key encryption mechanism in order to provide a greater degree of authentication.

Appendix A Concept Index

A

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Appendix B Function Index

C

F

H

O

Appendix C Variables Index

Appendix D Program Index

Appendix E Data Types Index

B

C

H

I

O

P

T

The Odyssey Story

Odyssey started out as a (un)conscious tribute to an epic of our own generation— *2001: A Space Odyssey* by Arthur C. Clark. We soon realized the potential for bad jokes and puns from the book and the movie. Soon after, the star-child Bowman was born, and the rest . . .

The Odyssey Code Group (in alphabetical order):

- Bobby Bhattacharjee
- Ken Calvert
- Youngsu Chae
- Richard Liston
- Shashidhar Merugu
- Matt Sanders
- Ellen Zegura

Table of Contents