NodeOS Interface Specification

AN Node OS Working Group

November 30, 2001

1 Introduction

This document defines the interface to the operating system running on each node (router) in an active network. A companion document describes the overall architecture of an active network [7]. That document identifies three layers of code running on each active node. At the lowest level, an underlying operating system (NodeOS) multiplexes the node's communication, memory, and computational resources among the various packet flows that traverse the node. At the next level, one or more execution environments (EE) define a particular programming model for writing active applications. To date, several EEs have been defined, including ANTS [16, 15], PLAN [1, 5], and CANES [3]. At the topmost level are the active applications (AA) themselves.

The goal of active networks is to make the network as programmable as possible, while retaining enough common interfaces so that active applications injected into the network can run on as many nodes as possible. In this context, it is not obvious where to draw the line between the EEs and the NodeOS. One answer is that there is no line: a single layer implements all the services required by the active applications. This is analogous to implementing a language runtime system directly on the hardware, as some JavaOSs have done. However, separating the OS from the runtime system makes it easier for a single node to support multiple languages. It also makes it easier to port any single language to many node types. This is exactly the rationale for defining a common NodeOS interface.

Deciding to separate the NodeOS and the EEs is only the first step; the second step is to decide where the EE/NodeOS boundary should be drawn. Generally speaking, the NodeOS is responsible for multiplexing the node's resources among various packet flows, while the EE's role is to offer AAs a sufficiently high-level programming environment. This is loosely analogous to the distinction between an exokernel and an OS library [8]. Beyond this general goal, the NodeOS interface is influenced by the following three high-level design goals:

1. The interface's primary role is to support packet forwarding, as opposed to running arbitrary computations. As a consequence, the interface is designed around the idea of network packet flows [4]: packet processing, accounting for resource usage, and admission control are all done on a per-flow basis. Also, because network flows can be defined at different granularities—e.g., port-to-port, host-to-host, per-application—the interface cannot prescribe a single definition of a flow.

2. We do not assume that all implementations of the NodeOS interface will export exactly the same feature set—some implementations will have special capabilities that EEs (and AAs)

may want to take advantage of. The interface should allow access to these advanced features. One important feature is the hardware's ability to forward certain kinds of packets (e.g., non-active IP) at very high speeds. To paraphrase, packets that require minimal processing should incur minimal overhead. A second feature is the ability to extend the underlying OS itself, i.e., extensibility is not reserved for the EEs that run on top of the interface. The NodeOS interface must allow EEs to exploit these extensions, but for reasons of simplicity, efficiency, and breadth of acceptable implementations, the NodeOS need not provide a means for an EE to extend the NodeOS directly. Exactly how a particular OS is extended is an OS-specific issue.

3. Whenever the NodeOS requires a mechanism that is not particularly unique to active networks, the NodeOS interface should borrow from established interfaces, such as POSIX.

In addition to these high-level design goals, specific details of the interface were influenced by our experience with three different NodeOS implementations [11].

2 Abstractions

The NodeOS interface defines five primary abstractions: *thread pools*, *memory pools*, *channels*, *files*, and *domains*. The first four encapsulate a system's four types of resources: computation, memory, communication, and persistent storage. The fifth abstraction, the domain, is used to aggregate control and scheduling of the other four abstractions. This section motivates and describes these five primary abstractions explaining the relationships among them in detail, and mentions the other abstractions that the NodeOS provides: events, the heap, packets and time.

2.1 Domains

The domain is the primary abstraction for accounting, admission control, and scheduling in the system. Domains directly follow from our first design decision: each domain contains the resources needed to carry a particular packet flow. A domain typically contains the following resources (Figure 1): a set of channels on which messages are received and sent, a thread pool, and is associated with a particular memory pool. Active packets arrive on an input channel (*inChan*), are processed by the EE using threads and memory allocated to the domain (dotted arc), and are then transmitted on an output channel (*outChan*).

Note that a channel consumes not only network bandwidth, but also CPU cycles and memory buffers. The threads that shepherd messages across the domain's channels come from the domain's thread pool and the cycles they consume are charged to that pool. Similarly, the I/O buffers used to queue messages on a domain's channels are allocated from (and charged to) the domain's memory pool. In other words, one can think of a domain as encapsulating resources used across both the NodeOS and an EE on behalf of a packet flow, similar to resource containers [2] and Scout paths [13]. A domain is not strictly a "user level" entity like a Unix process.

A given domain is created in the context of an existing domain, making it natural to organize domains in a hierarchy, with the root domain corresponding to the NodeOS itself. Figure 2 shows a representative domain hierarchy, where the second level of the hierarchy corresponds to EEs and domains at lower levels are EE-specific. In this example, the EE implemented in Domain A has

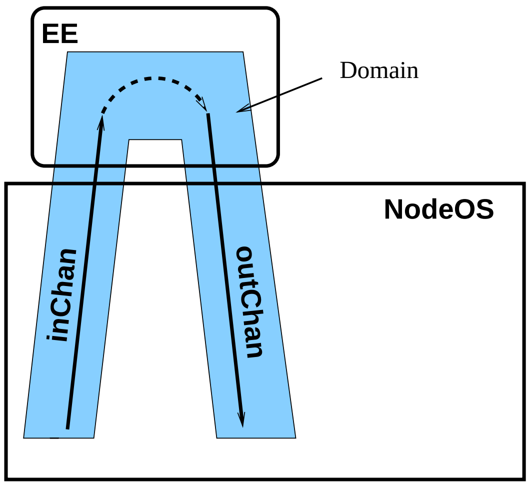

Figure 1: A domain consists of channels, memory, and threads needed for EE-specific processing.

chosen to implement independent packet flows in their own domains (Domain C through Z), while the EE running in Domain B aggregates all packets on a single set of channel, memory, and thread resources. The advantage of using domains that correspond to fine-grained packet flows—as is the case with the EE contained in Domain A—is that the NodeOS is able to allocate and schedule resources on a per-flow basis. (Domain A also has its own channels, which might carry EE control packets that belong to no specific sub-flow.)

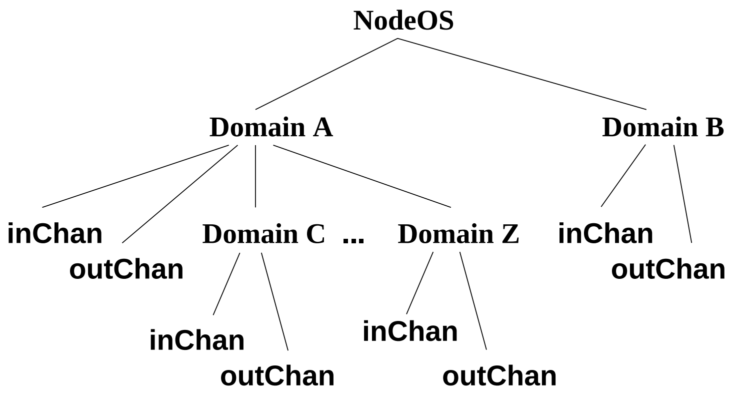

Figure 2: Domain hierarchy.

The domain hierarchy is used solely to constrain domain termination. A domain can be terminated by the domain itself, by the parent domain that created it, or by the NodeOS because the domain has violated some resource usage policy. Domain termination causes the domain and all its children to terminate, the domain's parent to be notified, and all resources belonging to the terminated domains are returned to the NodeOS.

Each parent domain contains a handler that is invoked just before a child domain is terminated by the NodeOS. This "imminent termination" handler allows the parent domain (generally running the EE) to reconcile any state it may have associated with the dying domain and free any resources it may have allocated on the child domain's behalf. The handler is invoked in the context of a thread in the parent domain; thus the parent domain pays for cleaning up an errant child domain. The handler is given a small, fixed amount of time to complete its cleanup. If the thread exceeds this limit, it, and the domain in which it runs, are terminated.

In contrast to many hierarchical resource systems (e.g., stride CPU schedulers [14]), the domain hierarchy is independent of resource allocation. That is, each domain is allocated resources according to credentials presented to the NodeOS at domain creation; resources given a child domain are not deducted from the parent's allocation, and resources belonging to a child domain are not returned to the parent domain when the child terminates. This design was based on the observation that requiring resources to be allocated in the same hierarchical manner as domains results is an overly restrictive resource model. For example, suppose an ANTS EE runs in a domain and creates new (sub)domains in response to incoming code capsules. These new domains should be given resources based solely on their credentials. They should not be restricted to some subset of the ANTS EE's resources, which they would be if resources followed the domain hierarchy.

2.2 Thread Pool

The *thread pool* is the primary abstraction for computation. Each domain contains a single thread pool that is initialized when the domain is created. Several parameters are specified when creating a thread pool, including the maximum number of threads in the pool, the scheduler to be used, the cycle rate at which the pool is allowed to consume the CPU, the maximum length of time a thread can execute between yields, the stack size for each thread, and so on.

Because of our decision to tailor the interface to support packet forwarding, threads execute "end-to-end"; that is, to forward a packet they typically execute input channel code, EE-specific code, and output channel code. Since a given domain cuts across the NodeOS and an EE, threads must also cut across the NodeOS/EE boundary (at least logically). This makes it possible to do end-to-end accounting for resource usage. Note that from the perspective of the NodeOS interface, this means that the thread pool primarily exists for accounting purposes. Whether or not a given NodeOS pre-allocates the specified number of threads is an implementation issue. Moreover, even if the NodeOS does pre-allocate threads, these threads may not be able to handle all computation that takes place on behalf of the thread pool; for example, they may not be allowed to run in supervisor mode. Any thread running on behalf of the thread pool, no matter how its implemented, is charged to the pool.

The fact that a thread pool is initialized when a domain is created, and threads run end-to-end, has two implications. First, there is no explicit operation for creating threads. Instead, threads in the pool are implicitly activated, and scheduled to run, in response to certain events, such as message arrival, timers firing, and kernel exceptions. Second, there is no explicit operation for terminating a thread. Should a thread misbehave—e.g., run beyond its CPU limit—the entire domain is terminated. This is necessary since it is likely that a thread running in an EE has already executed channel-specific code, and killing the thread might leave the channel in an inconsistent state.

As just described, threads are short-lived, "data driven" entities with no need for explicit identities. While this is sufficient for many environments, we expect some EEs to require "system" threads that are long-lived and not associated with any particular packet flow. For example, a JVM-based

EE might have a global garbage collection thread that, when it runs, needs to first stop all other threads until it is done. To support these environments, the API defines a small set of ***pthread***-inspired operations for explicit thread manipulation: sending an interrupt, blocking and unblocking interrupts, changing a scheduler-interpreted priority value, and attaching thread-specific data.

2.3 Memory Pool

The *memory pool* is the primary abstraction for memory. It is used to implement packet buffers (see Section 2.4) and hold EE-specific state. A memory pool combines the memory resources for one or more domains, making those resources available to all threads associated with the domains. Adding domains to a pool increases the available resources while removing domains decreases the resources. The amount of resources that an individual domain can contribute to a pool is either embodied directly in the domain's credentials or explicitly associated with the domain at creation time. The many-to-one mapping of domains to memory pools accommodates EEs that want or need to manage memory resources themselves.

To see this, consider an alternative approach in which the NodeOS enforces memory limits at domain granularity. In such a model, destroying a domain would require freeing the specific pages of memory that were explicitly allocated to that domain. Unfortunately, this approach is too fine-grained for a JVM-based EE in which sharing memory between domains is common. This would tightly constrain what the JVM could place in those pages. Though the JVM could easily ensure that objects created by a domain were placed in the correct memory, those objects could have incoming and outgoing references from/to objects in other domains. A domain's memory might also contain JITed code that is shared with other domains. Either of these situations could cause problems when the domain is destroyed and the memory freed. Attempting to avoid these scenarios or cleaning up after them is problematic. In contrast, in the mempool-based model, the JVM would still create domains as necessary, but the memory resources associated with each domain could be combined in a single mempool which the NodeOS would monitor. Although destroying a domain still requires reclaiming pages of memory from the JVM, it gives the JVM some flexibility in choosing which pages it will return. Note that per-domain limits can still be enforced, they are just enforced by the EE now rather than the NodeOS.

Memory pools have an associated callback function that is invoked by the NodeOS whenever the resource limits of the pool have been exceeded (either by a new allocation or by removing a domain from the pool). The callback function is registered when a memory pool is created by an EE. The NodeOS relies on the EE to release memory when asked; i.e., the NodeOS detects when a pool is over limit and performs a callback to the EE to remedy the situation. If the EE does not free memory in a timely manner, the NodeOS terminates all the domains associated with the pool. The rationale for these semantics is similar to that for domain termination give above: the EE is given a chance to clean up gracefully, but the NodeOS has fallback authority.

Allocation from memory pools is performed via a familiar **malloc**-style interface. While the granularity of memory allocation and accounting is implementation specific, there are no granularity constraints on the visible interface. A more complicated **mmap**-style interface was considered, but has been deferred to a future version of the specification.

Memory pools are independent, having no explicit or implicit relationship with other pools. A hierarchical arrangement, allowing constrained sharing, was considered but has been deferred to a future version of the specification.

Finally note that memory pools are primarily for resource accounting. Their relationship to to protection domains or address spaces is undefined. For example, an implementation may choose to place all memory pools in the same address space or it might map memory pools one-to-one with unique address spaces.

2.4 Channels

Channels are the primary abstraction for communication flows. Domains create channels to send, receive, and forward packets. Some channels are *anchored* in an EE; anchored channels are used to send packets between the execution environment and the underlying communication substrate. Anchored channels are further characterized as being either *incoming* (*inChan*) or *outgoing* (*outChan*). Other channels are *cut-through* (*c*utChan), meaning that they forward packets through the active node—from an input device to an output device—without being intercepted and processed by an EE. Clearly, channels play a central role in supporting our flow-oriented design. We crystallize this role at the end of this subsection; first we describe the various types of channels in more detail.

When creating an *inChan*, a domain must specify several things: (1) which arriving packets are to be delivered on this channel; (2) a buffer pool that queues packets waiting to be processed by the channel; and (3) a function to handle the packets. Packets to be delivered are described by a protocol specification string, an address specification string, and a demultiplexing (demux) key. The buffer pool is created out of the domain's memory pool. The packet handler is passed the packet being delivered, and is executed in the context of the owning domain's thread pool.

When creating an *outChan*, the domain must specify (1) where the packets are to be delivered and (2) how much link bandwidth the channel is allowed to consume (guaranteed to get). Packet delivery is specified through a protocol specification string coupled with an address specification string. The link bandwidth is described with an RSVP-like QoS spec [17].

Cut-through channels both receive and transmit packets. A *c*utChan can be created by concatenating an existing *inChan* to an existing *outChan*. A convenience function allows an EE to create a *cutChan* from scratch by giving all the arguments required to create an *inChan*/*outChan* pair. Cut-through channels, like input and output channels, are contained within some domain. That is, the processor cycles and memory used by a *c*utChan are charged to the containing domain's thread and memory pools, respectively. Figure 3 illustrates an example use of cut-through channels, in which "data" packets might be forwarded though the cut-through channel inside the NodeOS, while "control" packets continue to be delivered to the EE on an input channel, processed by the EE, and sent on an output channel.

The protocol specifications for *inChan*s and *outChan*s refer to the corresponding protocol modules built into the NodeOS. For example, "ipv4", "udp", or "anep". Components of the protocol specification string are separated by the *'/'* character. Included at one end of a protocol specification is the interface on which packets arrive or depart. Thus, a minimal specification is "if" (for all interfaces) or "if*N*" where *N* is the identifier of a specific virtual interface. For example "if0/ipv4/udp/anep" specifies incoming ANEP packets tunneled through IP version 4 on an *inChan*, while "ipv4/if" specifies outgoing IPv4 packets on an *outChan*. *cutChan* protocol specification strings have an identical syntax with the insertion of a '|' symbol to denote the transition from incoming packet processing to outgoing packet processing; for example, "ipv4/udp|udp/ipv4".

The address specification string defines destination addressing information (e.g., the destination UDP port number). The format of the address is specific to the highest level protocol in the protocol

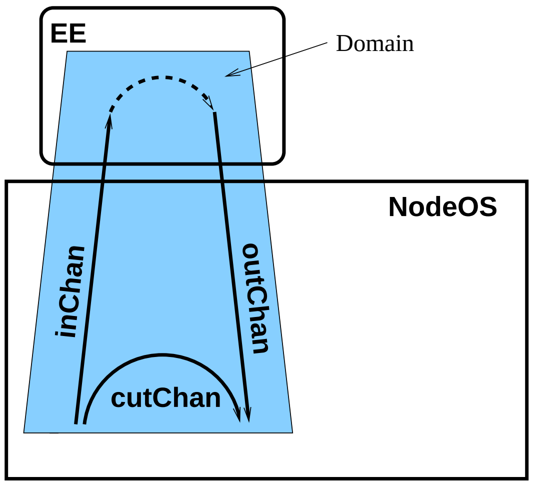

Figure 3: A domain with a cut-through channel.

specification (e.g., describing UDP addresses).

Simply specifying the protocol and addressing information is insufficient when an EE wants to demultiplex multiple packet flows out of a single protocol, such as a single UDP port. The demux key passed to the *inChan* specifies a set of 4-tuples: offset, length, value, mask. These tuples are compared in the obvious way to the payload of the protocol — payload being the non-header portion of the packet for a given protocol specification. For example, with a raw “if” specification, the payload is everything after the physical headers; with an “if/ipv4/udp” specification the payload is the UDP payload. Convenience functions are provided for creating filters that match well-known headers.

Note that demux keys and protocol specifications logically overlap. The difference lies in how NodeOS processes the packets. For example, an EE can receive UDP port 1973 packets by creating an *inChan* with a protocol of “if0” and demux key that matches the appropriate IP and UDP header bits, or by creating an *inChan* with a protocol of “if0/ipv4/udp”. The important and critical distinction is that the former case will not catch fragments at all, while the latter will perform reassembly and deliver complete UDP packets. Additionally, the former will provide the IP and UDP headers as part of the received packet whereas the latter will not.

We conclude our description of channels by revisiting our design goals. First, it is correct to view channels and domains as collectively supporting a flow-centric model: the domain encapsulates the resources that are applied to a flow, while the channel specifies what packets belong to the flow and what function is to be applied to the flow. The packets that belong to the flow are specified with a combination of addressing information and demux key, while the function that is to be applied to the flow is specified with a combination of protocol module names, such as “if0/ipv4/udp” and the handler function.

Second, cut-through channels are primarily motivated by the desire to allow the NodeOS to forward packets without EE or AA involvement. Notice that a *cutChan* might correspond to a standard forwarding path that the NodeOS implements very efficiently, perhaps even in hardware, but it might also correspond to a forwarding path that includes an OS-specific extension. In the former case, the EE that creates the *cutChan* is able to *control* the channel's behavior, similar to the control allowed by APIs defined for programmable networks [6, 9]. In the latter case, the EE that creates the *cutChan* is able to *name* the extension (for example, "if0/ipv4/extension/if1") and specify parameters according to a standard interface. However, exactly how this extension gets loaded and its interface to the rest of the kernel is an OS-specific issue; the NodeOS interface does not prescribe how this happens. In other words, cut-through channels allow EEs to exploit both performance and extensibility capabilities of the NodeOS.

2.5 Files

Files are provided to support persistent storage and coarse-grained sharing of data. The file system interface loosely follows the POSIX 1003.1 specification, and is intended to provide a hierarchical namespace to EEs that wish to store data using a file oriented interface. While not every NodeOS will have the ability to provide persistent storage, each NodeOS is encouraged to provide non-persistent storage based on the proposed interface.

We consider access control to be outside the scope of the NodeOS spec at this time. There are several interface calls that take an *Access Control Descriptor* argument, partly because their POSIX counterparts take a mode argument. This argument has been changed to an an_ACDdatatype, which only the NodeOS can operate on. This datatype is not yet defined.

2.5.1 Name Space

Each EE sees a distinct view of the persistent filesystem, rooted at a directory chosen at configuration time. In other words, "/" for the ANTS EE will be rooted at /ANTS, which implies that the only files that can be accessed by the ANTS EE reside in /ANTS. This insulates EEs from each other with respect to the persistent filesystem namespace. EE filesharing is not provided at this time.

However, in order to accommodate environments for which EE file sharing is desirable, future versions will provide an interface to allow EEs to access the entire namespace. This interface will most likely be either a function of NodeOS configuration parameters, or perhaps the security and resource credentials with which the EE is instantiated.

2.6 Other Abstractions

Apart from the five primary abstractions outlined above, the NodeOS API needs to provide abstractions for *events*, the *heap*, *packets* and *time* to the EE. The *event* abstraction allows a Domain to schedule an asynchronous event in the future, to be handled by a specific event handler. On behalf of an EE, the domain can schedule, detach and cancel the event.

The NodeOS also implements a *heap* for memory management, and provides the EE with an interface to allocate and free memory to and from the heap. This allows EEs to delegate memory management to the NodeOS if they so desire.

A *packet* encapsulates the data that traverses a channel. Packets are essentially a buffer-like structure built for fast adding and deleting of headers. Each packet is associated with only one domain for its lifetime.

EEs need some notion of time. Hence the NodeOS needs to provide a *time* as an abstraction to enables the EE to access the time of day, as viewed by the NodeOS.

2.7 Abstraction Summary

Four of the primary abstractions, the *thread pools, memory pools, channels* and *files*, encapsulate a system's four types of resources: computation, memory, communication, and persistent storage, respectively. The fifth abstraction, the domain, aggregates‘ control and scheduling of the other four abstractions. However, semantically, a particular domain does not necessarily subsume the instantiations of each of the other abstractions, or all of their components. Rather, its relationship to the other abstractions is is summarized in Figure 4, where we observe many-to-one, one-to-one and one-to-many mappings.

$$
\begin{array}{lcl}
\text{Domain} & \xrightarrow{M:1} & \text{Memory Pool} \\
\text{Domain} & \xrightarrow{1:1} & \text{Thread Pool} \\
\text{Domain} & \xrightarrow{1:M} & \text{Channel} \\
\text{Domain} & \xrightarrow{M:1} & \text{File Name Space}
\end{array}
$$

Figure 4: Relationship of Domains to the Memory, Thread, Channel and Persistent Storage Abstractions

A domain is the abstraction that contains the resources to carry a particular packet flow. Domains are created and terminated by domain creation and domain termination functions, respectively. Domains are organized hierarchically with respect to the creations and termination of its children. This is in contrast to domain resource allocation, which is non-hierarchical. Rather, a child domain is allocated memory, threads and channels based solely on the child's credentials, and upon termination, these resources are not returned to the parent domain.

The domain's many-to-one relationship with the primary memory abstraction, the memory pool, highlights the fact that EE domains may share memory. Memory pools are also organized hierarchically in terms of access control, and are created and terminated by memory pool creation and termination functions, respectively.

Each domain has a unique thread pool, the primary abstraction for computation. This one-to-one relationship is further emphasized by the fact that thread pools are created and terminated solely by means of domain creation domain termination funtions, respectively. Threads are expected to logically execute end-to-end, from handling incoming packets, to sending outgoing packets. Between the two, the thread my cross to the EE for further processing, or remain within the NodeOS if the cut-through path was invoked.

An EE may decide to create (or destroy) any number of channels within one domain, by invoking the channel creation (or channel termination) functions. Channels can be incoming – *inChan*s, outgoing – *o*utChans, or cut-through – *cutChan*s.

Files are the primary abstraction for persistent storage, and are created and removed in the usual way, via file open and file unlink operations, respectively. The file system is hierarchical, as most file systems are, and loosely follows the POSIX 1003.1 specification. Any number of domains within the same EE may access the same file namespace. Sharing the name space across EEs is currently not allowed.

The NodeOS provides additional "auxiliary" abstractions: events, to enable asynchronous scheduling of systems events in the future, a heap for memory management, packets to encapsulate the data traversing a channel, and the notion of time.

3 Interface

This section defines the interface for the five primary abstractions: domains, thread pool, memory pool, channels and files. Furthermore, we define additional miscellaneous interfaces provided by the NodeOS as abstractions for events, a heap, packets, time, network functions and a mechanism for bootstrapping the EE.

3.1 Conventions

Before detailing the specifics of the NodeOS API, we present the conventions used in the specification.

3.1.1 Symbol Namespaces

Nodeos implementations reserve, at least, the prefixes “an_”, “ani_”, “nodeos_”, “AN_”, and “NODEOS_” for visible symbols. Each implementation will probably reserve more prefixes. For example, the Moab implementation additionally reserves “moab_” and “ oskit_”.

All publicly visible symbols in the C function namespace, the C type namespace, and the C preprocessor macro namespace will fall within these defined prefixes.

3.1.2 A Word on Notation

To improve readability, this document employs four different style fonts in the text, for type names, variable names, function names and constants. Additionally, yet another style is used to highlight snippets of C code.

Type names are styled in an upright sans serif. This applies to standard C types and NodeOS types. For example, int, void *, an_Domain, an_Error.

Variable names are in sans serif italics. Examples are *domain*, *mem*, *lock*.

Function names have a boldface sans serif font. **an_domainCreate**, **an_mempoolFree** are some examples.

Constants are in small caps sans serif, with AN_DOMAIN_SIZE and AN_MUTEX_SIZE, being some examples.

Fragments of code look as follows:

```
typedef struct an_FooSpec {
    int          field1;
    an_Object    field2;
} * an_FooSpec;
```

3.1.3 Specifications and Objects

There are two kinds of structure involved in NodeOS API calls, *specifications* and *objects*. All the specifications have "Spec" in the type name, all other structures are objects. Specifications are structures with visible, well-defined fields. Objects are opaque structures. The most important distinction is that, from the perspective of the NodeOS, the lifetime of a specification is a single API call, while the lifetime of an object is from its explicit creation, via an API "create" call, until its explicit destruction, via an API "destroy" call. Note that standard C parameters, such as char * or void *, are not part of this convention.

As a result of these semantics, which are further clarified below, objects can be subcomponents of a specification, but specifications cannot be subcomponents of objects.

From the EE's perspective, it must ensure that the specification and any storage referenced by the specification are not mutated or reclaimed until the NodeOS API function returns. This enables the EE to pass the same specification to multiple APIs calls simultaneously.

From the perspective of the NodeOS, when a reference to a specification is passed as a parameter to a NodeOS API function, the NodeOS may freely access the specification and any storage that is referenced directly or indirectly by the specification. For example, if the specification contains a "char *" to a null-terminated string, the NodeOS may access the string storage. The documented fields of the specification structure will not be modified by the NodeOS; this guarantees that when the NodeOS API function returns, the EE will be able to locate pointers to storage that it placed into the specification, and which also enables the EE to pass it to multiple API calls at the same time. After a NodeOS API function returns, the NodeOS will retain no references to any specification or its subparts, except references that exist because either (1) a subcomponent is an object, or (2) the specification or one of its subcomponents is currently involved in another currently-running NodeOS API function call.

For example, when creating a domain object via **an_domainCreate**, consider the an_Domain parameter and the an_ThreadPoolSpec parameter. The domain parameter is an opaque object, whose memory cannot be reused until the domain is destroyed. The an_ThreadPoolSpec parameter's memory, however, can be reused immediately after the **an_domainCreate** function returns.

3.1.4 Arguments to the API Function Calls

There are four types of arguments passed to the functions of the API:

1. standard C types, such as void *, char *
2. pointers to functions, such as void *(*f)*(void * *arg*), or an_NotifyFunc
3. objects, such as an_Domain
4. specifications, such as an_ThreadPoolSpec

It is noteworthy that as arguments to an API call, or the entity returned by a function, both specifications and objects are passed by reference, as pointer type arguments. A template for declaring a specification type and an opaque object type is shown below.

```
typedef struct an_GenericSpec {
    type1     t1;
    type2     t2;

          ⋮

} * an_GenericTypeSpec;


typedef struct an_GenericObject {
    type1     t1;
    type2     t2;

          ⋮

} * an_GenericTypeObject;
```

The similarity in how both specification and object types are defined is intentional. The difference lies not in *how* they are defined, but rather in *where* they are defined. Specification structures need to be defined in the EE, and are therefore explicitly defined in this document. Object structurs, on the other hand, are defined in the NodeOS, since they need to remain implementation dependent and as such opaque to the API. This necessitates the EE to declare a pointer type of the form

```
typedef struct an_GenericObject * an_GenericTypeObject
```

in order to make API calls with objects as arguments. Such an arrangement provides three advantages. First, this allows for a set of portable header files, shared among all EEs, where all object pointer types are declared.[1] Second, this preserves the opaqueness of the objects from the EE's standpoint, thus allowing for different implementations of the NodeOS. And finally, as certain object types become better understood, such as an_Credentials, migrating them into specifications would be a simple matter of adding their structure definition to the EEs' common header files.

3.1.5 Implicit and Explicit Object Allocation

The NodeOS API is designed to support two general patterns of use. The first, and most common, is a model in which the EE does not care about tracking the memory used by the NodeOS to manage objects. Such an EE may pass NULL as the memory argument to all create calls, letting the NodeOS allocate the memory used to represent the object. These implicitly allocated objects will be automatically freed by the NodeOS when the corresponding destroy call is made. Furthermore, in this model the EE does not need to create and destroy threads at all. By specifying the implicit thread type and providing a "maximum number of threads" value in the thread pool specification used at domain creation time, thread objects will be automatically allocated and deallocated by the NodeOS as needed.

In the second model, an EE may wish to explicitly manage the memory used by the NodeOS for EE-created objects. In this world, the EE will pass an appropriately-sized chunk of memory to

[1] Such header files would also include constants specifiying the object sizes, as outlined in section 3.1.5.

all object create calls. The NodeOS will use the provided memory for the object, returning the a pointer to the object, which is essentially a pointer to that memory as the object handle returned by the create call. When such an explicitly allocated object is destroyed, the NodeOS will teardown any internal state associated with the object and stored in the object memory, but will *not* free the memory. When creating domains in the explicit-allocation model, the EE should first create thread objects and pass those objects to the NodeOS in the thread pool specification.

Note that the explicit-memory allocation model has some limitations. First, it is most effective in an implementation where the EE and NodeOS are tightly coupled (the so-called "trusted EE" model). If there is a protection boundary between the NodeOS and EE, then the NodeOS will likely still need to allocate its own private memory to hold the object state and will use the EE-provided memory to hold a reference to the actual object. Second, there is still memory allocated by the NodeOS that is not directly visible or controlable by the EE; for example, the actual memory used to hold packets.

3.2 Domains

an_Domain **an_domainCreate(**void * *mem*, an_Credentials *c*, an_ThreadPoolSpec *tp*, an_-MemPoolSpec *vm*, an_DomainInitFunc *initfunc*, void * *initarg*, an_DomainTermFunc *termfunc* **):**

Create a new domain using the specified credentials, thread pool and memory resources. The new domain is a child of the creating domain in the termination hierarchy. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_DOMAIN_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error. The form and meaning of the credentials are defined by the Security architecture [12, 10]. When the NodeOS has finished setting up the internal state of the new domain, one of the domain's threads is used to asynchronously invoke the provided intialization functions with the single argument *initarg*. Since this call is asynchronous, the create call may return before *initfunc* completes or is even started. The NodeOS does a synchronous callback to function *termfunc* in the parent domain as the first step in a child domain termination. The signature of an_DomainTermFuncis implementation specific but will always include a pointer to the domain being destroyed as the first argument.

an_ThreadPoolSpecand an_MemPoolSpecspecify how the thread and memory resources, respectively, are to be initialized.

```
typedef struct an_ThreadPoolSpec{
    /* description of threads */
    enum { AN_TH_IMPLICIT, AN_TH_EXPLICIT } ttype;
    union {
        /* AN_TH_IMPLICIT */
        struct {
            int    maxthreads;
            size_t stacksize;
        } im;
        /* AN_TH_EXPLICIT */
        struct {
            int    nthreads;
            an_Thread *threads;
        } ex;
    } threads;

    /* scheduler params */
    int            rate;
    an_Timespec    slice;
    an_Schedule    *sched;
} * an_ThreadPoolSpec;
```

Specifies the number of threads that can run concurrently in the domain and their scheduling parameters. Threads are specified either implicitly (*threads.im*) as a number of threads (*maxthreads*) and a per-thread stack size (*stacksize*), or explicitly (*threads.ex*) as a list of pre-created thread objects (*nthreads* and *threads*). Threads in the pool get *rate* cycles-per-second of normalized CPU bandwidth and are managed by scheduler *sched*. The *slice* parameter specifies a maximum, in seconds and nanoseconds, of CPU time that threads in this pool may run before yielding the processor (or exiting); if a thread runs beyond this limit, the NodeOS will terminate the entire domain. A value of zero specifies no time limit, threads in this pool will be scheduled preemptively.

```
typedef struct an_MemPoolSpec{
    int        nchunks;
    an_MemPool pool;
} * an_MemPoolSpec;
```

Specifies that the domain is associated with memory pool *pool* and is contributing *nchunks* chunks of memory to it. All allocations performed by the NodeOS on bahalf of the domain are charged to this pool. The chunk size is implemenatation specific, defined in bytes with the constant AN_NODEOS_MEMPOOLCHUNK_SIZE.

an_Error **an_domainDestroy(**an_Domain *domain* **):**

Destroy a domain *domain* and release all of its associated resources, including those held by children domains. Domain destruction is restricted by the domain hierarchy, that is, domains can destroy only themselves and their child domains. The per-domain *termhandler* will be invoked by the NodeOS on the target domain.

an_Domain **an_domainId(**void **):**

Returns the current domain ID.

an_Error **an_domainStartThread(**an_Domain *domain*, void *(*func)*(void * *arg*), void * *arg*, an_Thread* *retval* **):**

Start one of the domain's threads at function *func* with argument *arg*. If *retval* is non-zero, returns the ID of the started thread.

3.3 Thread Pool

an_Thread **an_threadId(**void **):**

Returns the current thread ID.

an_Error **an_threadStart(**an_Thread *thread*, an_Domain *domain*, void *(*func)(void *arg)*, void * *arg* **):**

Start a specific thread executing at function *func* with argument *arg* in domain *domain*.

void **an_threadExit(**void **):**

Return currently executing thread to the thread pool.

void **an_threadYield(**void **):**

Yield the processor from the current thread.

void **an_threadSleep(**const an_TimeSpec *delay* **):**

Delay the current thread for at least the given number of seconds and nanoseconds.

void **an_threadSetData(**void * *data* **):**

Set a per-thread pointer for the current thread.

void * **an_threadGetData(**void **):**

Return the current thread's per-thread data pointer. If **an_threadSetData** has not been previously called for this thread, the result of calling **an_threadGetData** is undefined.

void **an_threadInterrupt(**an_Thread *thread* **):**

Interrupt the specified thread and cause it to run an interrupt handler. If that thread is blocked in **an_condWait**, the interrupt handler is run without waiting for the an_Condto be signalled. If thread thread is engaged in another NodeOS operation, it is unspecified whether the interrupt handler runs before, during, or after, the nodeos operation.

The interface for setting a thread's interrupt handler, whether that handler is per-thread, per-domain, or global to the EE, and how the handler is invoked, are implementation-defined.

Caution: The interrupt routine can elect to terminate the thread, but terminating a thread executing an upcall from an *inChan* might leave the *inChan* in an inconsistent state.

void **an_threadBlockInterrupts(**void **):**

Block interrupts for the current thread. If **an_threadInterrupt** is called for this thread, the interrupt will be held until it calls **an_threadUnblockInterrupts** (or unblocks interrupts by some other implementation-defined means). A newly started thread has interrupts unblocked.

void **an_threadUnblockInterrupts(**void **):**

Unblock interrupts for the current thread. If **an_threadInterrupt** was previously called for this thread while its interrupts were blocked, that interrupt is delivered before **an_threadUnblockInterrupts** returns to its caller. It is unspecified whether posting multiple interrupts while blocked results in receiving multiple interrupts when unblocked, or just one.

an_Error **an_threadSetPrio(**an_Thread *t*, int *priority* **):**

Set the given thread's priority to the given value. Individual threads parameterize this policy with a single integer *priority* value. Although called a priority, the parameter has scheduler-specific semantics.

an_Error **an_threadGetPrio(**an_Thread *t*, u_int32_t * *retval* **):**

Get the priority associated with the given thread.

To provide synchronization between threads within a single domain hierarchy, two familiar abstractions are provided, mutual-exclusion locks and condition variables.

an_Mutex **an_mutexCreate(**void * *mem* **):**

Create a new an_Mutex object that can be used within this domain. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_MUTEX_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error. The new mutex is initially unlocked.

an_Error **an_mutex(**Destroy **):**an_Mutex *lock*

Destroy the given an_Mutex object. This call will unlock the an_Mutex as well as removing its internal state.

an_Error **an_mutex(**Lock **):**an_Mutex *lock*

Lock a mutex, blocking until it becomes available. Note that this thread might not run interrupt handlers until after it succeeds in acquiring the lock.

an_Error **an_mutex(**TryLock **):**an_Mutex *lock*

Attempt to lock the mutex. Returns no error if successful, or some error value if another thread already holds the lock.

Error values need to be defined.

an_Error **an_mutex(**Unlock **):**an_Mutex *lock*

Unlock a mutex previously locked by the calling thread.

an_Cond **an_condCreate(**void * *mem* **):**

Initialize a new *an_Cond* object that can be used within this domain The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_COND_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error.

an_Error **an_condDestroy(**an_Cond *condvar* **):**

Destroy the given an_Cond object. Destroying an an_Cond object with waiting threads will return an error.

an_Error **an_condWait(**an_Cond *condvar*, an_Mutex *mutex* **):**

The calling thread must hold the given mutex lock. Release the lock, and wait until **an_-condSignal** or **an_condBroadcast** is called on this *an_Cond* object. On return, the mutex lock is again held. There may be spurious wakeups. If **an_threadInterrupt** is called on this thread while it is blocked in **an_condWait**, the interrupt handler runs and then the thread resumes its block (or perhaps sees a spurious wakeup).

an_Error **an_condSignal(**an_Cond *condvar* **):**

If any threads are blocked in **an_condWait** on this *an_Cond* object, wake one of them.

an_Error **an_condBroadcast(**an_Cond *condvar* **):**

If any threads are blocked in **an_condWait** on this *an_Cond* object, wake all of them.

3.4 Memory Pool

an_MemPool **an_mempoolId(**void **):**

Get the mempool associated with the current thread's domain.

an_MemPool **an_mempoolCreate(**void * *mem*, an_MemPool *parent*, an_MemPoolFull *callback* **):**

Create a new mempool with no associated domains. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_MEMPOOL_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error. The *parent* parameter is reserved for future use, and should be NULL. The *callback* routine is invoked whenever the NodeOS detects that the indicated pool is over its memory limit. Parameters to the callback function include the pool, the current memory consumption of the pool (including the amount that puts it over limit) in AN_NODEOS_MEMPOOLCHUNK_SIZE chunks, and the memory limit of the pool in chunks. The callback function should use **an_mempoolFree** to free up memory.

an_Error **an_mempoolDestroy(**an_MemPool *pool* **):**

Destroy a memory pool. The indicated pool must not have any attached domains or the call will fail. The caller is responsible for freeing the memory occupied by the now defunct memory pool object.

void **an_mempoolAlloc(**an_MemPool *pool*, an_Size *size* **):**

Allocate a chunk of memory of at least the indicated size. The returned memory can be larger than the requested size if the implementation manages fixed-size chunks (for accounting reasons). Returns NULL if there are insufficient resources in the memory pool (and the pool's callback fails to return any).

void **an_mempoolFree(**an_MemPool *pool*, void * *mem*, an_Size *size* **):**

Free a chunk of memory previously allocated with **an_mempoolAlloc**.

3.5 Channels

3.5.1 Channel Creation

an_InChan **an_inchanCreate(**void * *mem*, an_Domain *domain*, an_DemuxKey *demuxKey*, char *protospec*, char *addrspec*, an_NetSpec *netspec*, an_ChanRecvFunc *deliverfunc*, void * *deliverarg* **):**

Creates an input channel. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_CHAN_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error. Only packets matching the given *demuxkey* are handed to the deliver function. Processing specified by the *protocolspec* argument is performed on the packet. The *d*emuxkey does not match against the headers used in the protocol processing. *protospec* must specify, at a minimum, an interface on which to receive packets. See Section 3.5.3 for a complete description of this specification. *addrspec* specifies the addresses to match. It is a restricted filter and its format is dependent upon the protocolSpec. See Section 3.5.4 for a description of this specification.

The *netspec* parameter is an instance of a an_NetSpec, which is used to describe the resources to be associated with a channel.

```
typedef struct an_netspec {
        int             maxthreads;
        unsigned int    bandwidth;
        struct {
                int     npbufs;
                anPacketBuffer*pbufs;
        }               buffers;
} an_netspec_t;
```

For input channels, the *maxthreads* field indicates the maximum number of concurrent threads that can be processing packets and *buffers* contains an array and count of buffers used for receiving incoming packets. The *bandwidth* field is ignored for input channels.

Issue: anPacketBuffer is not yet defined.

When a packet matches the an_DemuxKeyand has been processed as indicated by the protocol specification, it is delivered to the EE by invoking *deliverfunc* using a thread scheduled out of the owning domain's ThreadPool. *deliverarg* will be passed verbatim to *deliverfunc*. The exact type of *an_ChanRecvFunc* is left to the NodeOS implementation. It should include at least the *deliverarg* and a pointer to the packet data.

an_OutChan **an_outchanCreate(**void * *mem*, an_Domain *domain*, char * *protospec*, char * *addrspec*, an_NetSpec *netspec* **):**

Create an output channel on which packets can be sent. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_CHAN_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error.

The *protospec* and *addrspec* define the processing and headers that are attached to any packets sent on this channel. The *netspec* parameter is an instance of a an_NetSpec, as with input channels. For output channels, only the *bandwidth* field is used to describe the maximum bandwidth available to the channel.

Issue: The units of the bandwidth parameter are not yet defined.

an_CutChan **an_cutchanSplice(**void * *mem*, an_Domain *domain*, an_InChan *inchan*, an_OutChan *outchan*, char * *protospec* **):**

Create a cut-through channel that processes packets from the given *inchan* and pushes them through the processing specified by *protospec* and finally out the given *outchan*. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_CHAN_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error. Resources for cut-through channel processing are taken from the an_NetSpecparameters specified when creating the component input and output channels.

The *protospec* tells the NodeOS what sort of processing is done on the packets. E.g., "ipv4/udp/magicIn|magicOut/udp/ipv4". Generally, the *inchan* and *outchan* will only contain a single "if" module in their specification and the *cutchan* will only contain higher level modules. However, that is not required.

Note that the *inchan* provided must have its *an_deliverfunc* set to NULL. It is an error to send packets on an *outchan* that is associated with a *cutchan*.

an_CutChan **an_cutchanCreate(**void * *mem*, an_Domain *domain*, an_DemuxKey *demuxkey*, char * *inprotospec*, char * *outprotospec*, char * *inaddrspec*, char * *outaddrspec*, an_NetSpec *netspec* **):**

A convenience function that is equivalent to creating an *inchan*, a *outchan* and splicing them together. However, no handles on the underlying *inchan* or *outchan* are available. Resources for cut-through channel processing are given in *netspec*.

3.5.2 Using and Destroying Channels

Issue: we also need to define protocol-specific attributes that active protocols use to get/set parameters associated with passive protocols; e.g., report the interface that a given packet arrived on.

an_Error **an_outchanSend(**an_OutChan *outchan*, an_Packet *packet* **):**

Sends packet *packet* on output channel *outchan*. If the send call returns with no error indication, then the packet has been commited to the wire; that is, the channel send operation is synchronous.

A future version of this specification will include additional interfaces to allow asynchronous transmission of packets.

an_Error **an_inchanDestroy(**an_InChan *inchan* **):**

Destroy the given *inchan*. No more packets will be received. Active invocations of the channel deliver function are unaffected. If this *i*nchan was attached to a *cutchan* then the *cutchan* must be destroyed first.

an_Error **an_outchanDestroy(**an_OutChan *outchan* **):**

Destroy the given *outchan*. No more packets may be sent on this channel. If this *outchan* was attached to a *cutchan* then the *cutchan* must be destroyed first.

an_Error **an_cutchanDestroy(**an_CutChan *cutchan* **):**

Destroy the given *cutchan*. The *cutchan* must be destroyed before its *inchan* or *outchan* can be destroyed.

3.5.3 Protocol Specification

As outlined in section 2.4, a protocol specification string consists of the individual protocol components separated by "/"; for example, "if/ipv4/udp". Encapsulation is supported; for example, "if/ipv4/ipv4". The order (position) of the protocol component composition should make sense; that is, "if/udp/ip" is illegal, and the *an_chanCreate* function would return NULL for an error. Below are the protocol components that all NodeOS implementations will support, and their definitions and restrictions.

"if" : any network interface. This refers to a virtual interface to a hardware device provided by the NodeOS. Generally, only meaningful when combined with higher-level protocol components since there is no way to specify a meaningful address component when multiple interface types are present.

"if$n$" : a specific, also virtual, network interface n, where $n = 0, 1, \ldots$.

"ipv4" IPv4 level processing (fragmentation, reassembly). Must be preceded by "if[n]/".

"ipv6" IPv6 level processing (fragmentation, reassembly). Must be preceded by "if[n]/".

"tcp" : TCP processing (ACKs, timeouts, etc). Must be preceded by "ipv4/" or "ipv6/".

"udp" : UDP processing. Must be preceded by "ipv4/" or "ipv6/".

"anepv1" : ANEP version 1 processing. Must be preceded by "if[n]/" or "udp/".

"" : Every packet. The empty protocol spec is a promiscuous mode receive module. This module only makes sense on shared media and is only allowed on *inChan*s.

3.5.4 Address Specification

For each protocol component, the corresponding address format is provided. The "highest-level" protocol component in a chain determines the address format. As described in section 2.4, this format is identical for *inChan*s and *outChan*s, and is concatenated twice, with a "|" separator, for *cutChan*s.

"if" : "<srcmacaddr>:<dstmacaddr>[:<mac-specific-info>]"

Address is a MAC address in hex. Format is specific to the virtual interface device (e.g. for ethernet, 6 bytes). The optional MAC-specific field could include other, pertinent information from the link header that could not otherwise be matched by the demuxkey (e.g., for ethernet, the packet type).

Example 1: "0x0000e45907:0x0015006a00" (the "0x" prefix is optional).

Example 2: "0x0000e45907:0x0015006a00:0x800" matches only ETHERTYPE_IP packets.

"ipv4" : "<srcaddr>[:<destaddr>[:<protocol>]]"

<srcaddr> is a dotted quad or hostname. <protocol> is a number between 0 and 256, which refers to the higher level protocol to which this packet expects to be passed. <destaddr> is a dotted quad or hostname and must be a name for one of the interfaces on the local host. '*' can be used to skip any field. As is the case with IPv4, <destaddr> must be a name for one of the interfaces on the local host. '*' can be used to skip any field.

Example 1: "*:*:localnet" matches all IPv4 packets destined to the "localnet" interface on the router.

Example 2: "www.cs.utah.edu:*:*" matches all IPv4 packets from www.cs.utah.edu that are destined to this router.

Example 3: "18.34.100.80" matches all IPv4 packets destined to the receiving router from this specific (source) address.

"ipv6" : "<srcaddr>[:<destaddr>[:<protocol>]]"

<srcaddr> and <destaddr> are in the form of *x*:x:x:x:x:x:x:x, where "*x*" is a hexadecimal representation of a 16-bit piece of the address, or in the form of *x*:x:x:x:x:x:d.d.d.d, where "*d*" is a decimal digit, and the quad of these decimal digits is of the IPv4 format above. In essence, the addressing rules from RFC 1884 [] apply, such as dropping consecutive 0s and replacing the last 32 bits with an IPv4 address.

Example 1: "FEDC:BA98:7654:3210:FEDC:BA98:7654:3210" matches all IPv6 packets destined to the receiving router from this specific (source) address.

Example 2: "::FFFF:128.112.152.6" matches all IPv6 packets from that specific source address destined to the receiving router.

Example 3: *another example is needed here*

<protocol> can only be used if the *NextHeader* field of the packet's header, as specified in the IPv6 RFC 1883 [], is a valid protocol number from the IPv4 *Protocol* header field.

Example 4: "" *another example is needed here*

“udp” : “<srcaddr>[:<srcport>[:<destaddr>[:<destport>]]]”

<srcaddr> and <destaddr> are as in the “ipv[4|6]” spec address format.

<destport> and <srcport> are UDP port identifiers between 0 and 65535.

Example 1: “*:*:*:80” matches all UDP traffic on port 80 destined to this router.

Example 2: “18.34.100.80:*:localnet” matches all UDP traffic destined to this router from a specific machine that arrives on the ’localnet’ interface of this router.

“tcp” : “<srcaddr>[:<srcport>[:<destaddr>[:<destport>]]]”

<srcaddr> and <destaddr> are as in the “ipv[4|6]” spec address format.

<destport> and <srcport> are TCP port identifiers between 0 and 65535.

Example 1: “*:*:*:80” matches all TCP traffic on port 80 destined to this router.

Example 2: “18.34.100.34:*:localnet” matches all TCP traffic destined to this router from a specific machine that arrives on the ’localnet’ interface of this router.

“anepv1” : “<typeid>”

<typeid> is a 16-bit integer, as described in the ANEP RFC.

3.5.5 Demultiplex Keys

Demultiplex keys are used to filter incoming packets. See **an_inchanCreate** in Section 2.4. A demux key is logically a set of bytes that must be matched at various offsets in a packet. The NodeOS provides a clean, high-level interface to allow implementations to make low-level optimizations transparently.

Demux keys are applied to the payload of a channel packet, after protocol processing has occured and after the limited filtering of the address specification associated with a channel. For example, using a demux key to match an ethernet header only makes sense when applied to a “raw” channel.

Demux keys are built up incrementally using the various an_demuxkeyAdd functions. When adding a demux key “segment,” the offset at which it applies is implied as “immediately after the preceding segment.” That is, the first segment added is applied to offset zero of each packet’s payload while the second segment is applied starting *length* bytes into the payload, where *length* is the length of the first demuxkey segment, and so on.

One consequence of using implied offsets is the inability to build up demux keys involving protocols with variable length fields, that is, protocols where the offset of one field may be given as a value in the packet itself. Some common cases of variable length protocol data, such as IPv4 options, are handled by specialized an_demuxkeyAdd functions as described below.

Providing a more flexible packet filtering mechanism is left to NodeOS implementation specific extensions and may be standardized in a future revision of the specification. The current, simple matching scheme of a fixed offset, mask and length provides a “least common denominator” solution of general applicability and which lends itself to efficient implementations.

Issue: can a demux key be add’ed to after it has been associated with a channel?

Issue: can the same demux key be associated with multiple channels?

an_DemuxKey **an_demuxkeyCreate(** void * *mem*, an_Domain *domain* **):**

Create a new demux key. The *mem* parameter indicates how the NodeOS should obtain memory for the new object. If not NULL, *mem* must point to a block of memory of at least of size AN_DEMUXKEY_SIZE. If NULL, the NodeOS will allocate the memory itself on behalf of the current domain. A pointer to the resulting object is returned, or NULL if there was an error.

A newly created demux key, without any Add operations applied and when associated with a channel, will match any packet.

an_Error **an_demuxkeyDestroy(**an_DemuxKey *key* **):**

Destroy the given demux key. The demux key must not be associated with any channel.

an_Error **an_demuxkeyAdd(**an_DemuxKey *key*, u_int32_t *length*, u_int8_t * *sequence*, u_int8_t * *mask* **):**

Define a sequence of bytes that this demux key must match. The *sequence* will be compared to the packet at the current payload offset. At most *length* bytes will be compared. The *sequence* defines the array of bytes to compare. If *mask* is non-NULL, only bits that are active in the *mask* will be checked. The *mask* argument must be at least *length* bytes.

an_Error **an_demuxkeyAddEth(**an_DemuxKey *key*, int *flags*, u_int8_t *ethsrc[6]*, u_int8_t *ethdst[6]*, u_int8_t *etype* **):**

A specialized version of **an_demuxkeyAdd** used to construct a demux key segment matching an ethernet header. The *flags* parameter is used to indicate which of the remaining parameters are to be matched exactly. If the An_DEMUXKEY_ETH_ANYSADDR flag is not given, *ethsrc* contains the ethernet source address to match, otherwise any value is allowed. If the An_DEMUXKEY_ETH_ANYDADDR flag is not given, *ethdst* contains the ethernet destination address to match, otherwise any value is allowed. If the An_DEMUXKEY_-ETH_ANYETYPE flag is not given, *etype* contains the ethernet packet type to match, otherwise any value is allowed.

an_Error **an_demuxkeyAddIPv4(**an_DemuxKey *key*, int *flags*, u_int32_t *ipsrc*, u_int32_t *ipsrcmask*, u_int32_t *ipdst*, u_int32_t *ipdstmask*, u_int8_t *protocol* **):**

A specialized version of **an_demuxkeyAdd** used to construct a demux key segment matching an IPv4 header. Though it only compares against the fixed part of the IP header, it will correctly skip any IP options so that following key segments will be applied at the correct offset. This segment does *not* do a checksum comparison, thus it might match a corrupted IP packet. The *flags* parameter is used to indicate which of the remaining parameters are to be matched exactly. If the An_DEMUXKEY_IPV4_ANYSADDR flag is not given, *ipsrc* and *ipsrcmask* contain the source address and mask values to use, otherwise any value is allowed. If the An_DEMUXKEY_IPV4_ANYDADDR flag is not given, *ipdst* and *ipdstmask* contain the destination address and mask values to use, otherwise any value is allowed. If the An_DEMUXKEY_IPV4_ANYPROTO flag is not given, *protocol* contains the protocol type to match, otherwise any value is allowed.

an_Error **an_demuxkeyAddUDP(**an_DemuxKey *key*, int *flags*, u_int16_t *srcport*, u_int16_t *dstport* **):**

A specialized version of **an_demuxkeyAdd** used to construct a demux key segment matching a UDP header. This segment does *not* do a checksum comparison, thus it might match a corrupted UDP packet. The *flags* parameter is used to indicate which of the remaining parameters are to be matched exactly. If the An_DEMUXKEY_UDP_ANYSPORT flag is not given, *srcport* contains the source port to match, otherwise any value is allowed. If the An_DEMUXKEY_UDP_ANYDPORT flag is not given, *dstport* contains the destination port to match, otherwise any value is allowed.

an_Error **an_demuxkeyAddTCP(**an_DemuxKey *key*, int *flags*, u_int16_t *srcport*, u_int16_t *destport* **):**

A specialized version of **an_demuxkeyAdd** used to construct a demux key segment matching a TCP header. This segment does *not* do a checksum comparison, thus it might match a corrupted TCP packet. The *flags* parameter is used to indicate which of the remaining parameters are to be matched exactly. If the An_DEMUXKEY_TCP_ANYSPORT flag is not given, *srcport* contains the source port to match, otherwise any value is allowed. If the An_-DEMUXKEY_TCP_ANYDPORT flag is not given, *dstport* contains the destination port to match, otherwise any value is allowed.

an_Error **an_demuxkeyAddANEP(**an_DemuxKey *key*, int *flags*, u_int8_t *version*, u_int16_t *protoid* **):**

A specialized version of **an_demuxkeyAdd** used to construct a demux key segment matching an ANEP header. Though it only compares against the fixed part of the ANEP header, it will correctly skip any ANEP options so that following key segments will be applied at the correct offset. The *flags* parameter is used to indicate which of the remaining parameters are to be matched exactly. If the An_DEMUXKEY_ANEP_ANYVERSION flag is not given, *version* contains the version to match, otherwise any value is allowed. If the An_-DEMUXKEY_ANEP_ANYPROTOID flag is not given, *protoid* contains the protocol ID to match, otherwise any value is allowed.

3.6 Filesystem

For the Filesystem specification, the equivalent POSIX section numbers are given where appropriate.

an_Error **an_fileClose(**an_Domain *domain*, an_File *file* **):**

Close the open file referenced by *file*. See POSIX Section 6.3.1.

an_Error **an_fileFsync(**an_Domain *domain*, an_File *file* **):**

Cause all modified data and attributes of *file* to be moved to permanent storage. See POSIX Section 6.6.1.

an_Error **an_fileTruncate(**an_Domain *domain*, an_File *file*, an_Offset *length* **):**

Cause the file referenced by *file* to be truncated or extended to *length*. See POSIX Section 5.6.7.

an_Error **an_fileLseek(**an_Domain *domain*, an_File *file*, an_Offset *offset*, an_Whence *whence* **):**

Set the seek pointer for the file referenced by *file* to *offset*, according to the directive *whence*. See POSIX Section 6.5.3. The allowed values of whence are as follows:

AN_SEEK_SET: Offset is set to var offset bytes.

AN_SEEK_CUR: Offset is set to its current position plus var offset bytes.

AN_SEEK_END: Offset is set to the size of the file plus var offset bytes.

an_Error **an_fileMkDir(**an_Domain *domain*, const char * *path*, an_ACD *acd* **):**

The directory given by *path* is created with the access permissions specified by *acd*. See POSIX Section 5.4.1.

an_Error **an_fileRead(**an_Domain *domain*, an_File *file*, an_Size *nbytes*, void * *retval*, an_-Size * *retlen* **):**

Read *nbytes* of data from the file referenced by *file* into the buffer pointed to by *buf*. The actual number of bytes read is returned in *retlen*. See POSIX Section 6.4.1.

an_Error **an_fileRename(**an_Domain *domain*, const char * *from*, const char * *to* **):** Rename the file named *from* to new name var to. See POSIX Section 5.5.3.

an_Error **an_fileWrite(**an_Domain *domain*, an_File *file*, an_Size *length*, void * *buf*, an_Size * *retval* **):**

Write *nbytes* of data to the file referenced by *file* from the buffer pointed to by *buf*. The actual number of bytes written is returned in *retval*. See Section POSIX 6.4.2.

an_Error **an_fileUnlink(**an_Domain *domain*, const char * *path* **):**

Remove the file named by *path* from its directory. In a departure from POSIX semantics, if *path* is an empty directory, the directory is removed. See Section POSIX 5.5.1.

an_Error **an_fileOpen(**an_Domain *domain*, const char * *path*, an_Flag *flags*, an_ACD *acd*, an_File ** *retval* **):**

The file specified by *path* is opened for reading and/or writing, according to the *flags* argument. The new file object is returned in the location specified by var file. See POSIX Section 5.3.1. The *flags* are specified by or'ing an appropriate subset of the following values:

AN_O_RDONLY: open for reading only

AN_O_WRONLY: open for writing only

AN_O_RDWR: open for reading and writing

AN_O_APPEND: append on each write

AN_O_CREAT: create file if it does not exist

AN_O_TRUNC: truncate size to 0

AN_O_EXCL: error if create and file exists an_Error **an_fileStat(**an_Domain *domain*, const char * *path*, an_Stat *retval* **):** Obtain information about the file named by *path*, and store that information in the buffer pointed to by *retval*. See Section POSIX 5.6.2. The an_Statstructure is defined as follows:

```
typedef struct an_Stat {
  struct an_TimeSpec st_atimespec;  /* time of last access */
  struct an_TimeSpec st_mtimespec;  /* time of last
                                       modification */
  struct an_TimeSpec st_ctimespec;  /* time of last status
                                       change */
  an_Mode         st_mode;       /* inode protection mode */
  an_Offset       st_size;       /* file size, in bytes */
} * an_Stat;
/* st_mode */
#define        S_IFDIR  0040000  /* directory */
#define        S_IFREG  0100000  /* regular */
```

Caution: Maintenance of the atime and ctime fields of the stat structure are considered "optional" (may always return as zero), and are not required to be implemented by the underlying filesystem.

an_Error **an_fileFstat(**an_Domain* *domain*, an_File *file*, an_Stat *retval* **):** Obtain information about the open file referenced by *file*, and store that information in the buffer pointed to by *retval*. Otherwise, *f*stat behaves identically to the *stat* function. See POSIX Section 5.6.2.

3.7 Other Abstractions

There are a few other abstractions and library functions that the NodeOS needs to provide.

3.7.1 Events

The Event abstraction allows a Domain to schedule something to occur asynchronously in the future. Events are handled by event handlers.

*typedef void (*an_EventHandler)(an_Evente, void *arg);*

an_Error **an_eventSchedule(**EventHandler *f*, void * *arg*, u_long *t*, an_Event * *retval* **):**

Schedule EventHandler *f* to occur at least *t* microseconds in the future.

an_Error **an_eventDetach(**an_Event *e* **):**

Release the handle on the given event.

an_Error **an_eventCancel(**an_Event *e*, an_Result * *retval* **):**

Returns one of these values: AN_EVENT_FINISHED, AN_EVENT_RUNNING, or AN_-EVENT_CANCELED.

an_Error **an_eventIsCanceled(**an_Event *e* **):** Returns a boolean value that specifies whether or not the given Event has been canceled.

3.7.2 Packets

Packets encapsulate the data that traverses a channel. Packets are essentially a buffer-like structure built for fast adding and deleting of headers.

an_Packet **an_packetCreate(**void * *mem*, an_Domain *domain*, u_int8_t * *buf*, an_Size *len* **):** Create a packet with initial contents contained by *buf*. The *domain* designates the domain the packet will be associated with for its lifetime.

an_Error **an_packetDestroy(**an_Packet *p* **):** Destroy the named packet.

an_Error **an_packetPush(**an_Packet *p*, u_int8_t * *buf*, an_Size *len* **):** Push the data contained in *buf* onto the front of packet *p*.

an_Error **an_packetPop(**an_Packet *p*, an_Size *len*, u_int8_t *retval* **):** Remove the first *len* of data from packet *p* and return them.

an_Error **an_packetDuplicate(**an_Domain *domain*, an_Packet *p*, an_Packet *copy* **):** Creates a duplicate of packet *p*. The *domain* designates the domain the new packet will be associated with for its lifetime.

an_PacketWalk **an_packetwalkCreate(**void * *mem*, an_Domain *domain*, an_Packet *p* **):** Initializes context needed to walk (traverse) all the buffers in packet *p*. If not NULL, the block of memory pointed to by *mem* must have size at least AN_PACKETWALK_SIZE. The *domain* designates the domain to which any memory operations will be charged. Specifically the *context* may require memory allocation.

an_Error **an_packetwalkDestroy(**an_PacketWalk *context* **):** Tears down the PacketWalk context.

an_Error **an_packetwalkNext(**an_PacketWalk *context*, an_Size *retlen*, u_int8_t ** *retval* **):** Returns the next data buffer in the Packet associated with the given PacketWalk.

3.7.3 Time

Execution Environments need some notion of time, hence the NodeOS needs to provide a structure and one function:

```
typedef struct an_TimeSpec {
    uint32_t      tv_sec;
    uint32_t      tv_nsec;
} * an_TimeSpec;
```

an_Error **an_timeGetTime(**an_TimeSpec *t* **):** Fills in the current time of day. Though described in seconds and nanoseconds, the actual granularity of the system's clock is not specified.

3.7.4 Miscellaneous Network Functions

There are a few other miscellaneous network-related functions a NodeOS will need to provide to Execution Environments:

***uint32_t an_htonl(uint32_t hostlong)*:** Converts the given *hostlong* from host byte order to network byte order.

***uint16_t an_htons(uint16_t hostshort)*:** Converts the given *hostshort* from host byte order to network byte order.

***uint32_t an_ntohl(uint32_t netlong)*:** Converts the given *netlong* from network byte order to host byte order.

***uint16_t an_ntohs(uint32_t netshort)*:** Converts the given *netshort* from network byte order to host byte order.

***an_ModuleList getModuleInfo( )*:** Returns information about the available network modules, including device drivers.

3.7.5 Bootstrapping an Execution Environment

At boot time the NodeOS will create a root domain and call the user provided function *an_boot* from a thread owned by the root domain. In this manner control is passed to an Execution Environment.

AN_ROOTDOMAIN_THREADMAX: Macro that defines the maximum number of threads allowed in the root domain.

***void an_boot(void)*:** Function called by the NodeOS to bootstrap the root execution environment. This function executes in a thread owned by the root domain.

4 Future Issues

A number of issues were not addressed in the first version of this specification, which we expect future versions of the interface to incorporate.

For the purpose programming flexibility of the EE, the "event" interface in underspecified. A richer interface would include upcall functions to prompt for event notification, when a particular event occurs. Possibly, a mechanism tying channel operations and events would be desirable. A more fundamental issue would be whether to declare the event as a "first-class" citizen of the NodeOS interface; that is, should it be one of the primary abstractions? This remains an open question.

A major issue that needs to be addressed in forthcoming versions is a more explicit mechanism for authentication and resource allocation. *Steve Schwab?*

- general pattern matcher?
- new calls for asynchronous channels: the calls would signal/callback on completion

- A way to specify a variable offset for *an_demuxkeyMatch* would be useful. E.g. to define *offset* as the value of the 4-bit quantity starting at bit 4 of the header.
- add the capability to specify demuxing on specific ANEP options
- An API is needed to provide node management. Should it be possible to intercept all packets, then re-inject them into channels? This is related to the question of allowing a packet to match multiple channels. A paragraph should be written on this, perhaps referencing BBN.
-

5 Editor's Address

Comments should be sent to the mailing list:

`activenets_nodeos@ittc.ukans.edu`

or to the editor:

Larry Peterson
Department of Computer Science
35 Olden Street
Princeton, NJ 08544
Phone: +1.609.258.6077
Fax: +1.609.258.1771
Mail: `llp@cs.princeton.edu`

6 Acknowledgments

Many people have contributed to this document, including Mason Katz, Rob Piltz, Steve Schwab, Oliver Spatscheck, Godmar Back, Daniel Maskit, Jay Lepreau, Roland McGrath, Patrick Tullmann, Mike Hibler, Leigh Stoller, John Hartman, and Yitzchak Gottlieb.

Appendix A: Specifications
Appendix B: Objects

References

[1] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings of the ACM SIGCOMM '97 Conference*, pages 101–111, September 1997.

[2] Gaurav Banga, Peter Druschel, and Jeffrey Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symp. on Operating System Design and Impl.*, pages 45–58, February 1999.

[3] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. Congestion control and caching in CANES. In *ICC '98*, 1998.

[4] David Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of the SIGCOMM '88 Symposium*, pages 106–114, August 1988.

[5] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *ICFP 98*, pages 86–93, September 1998.

[6] IEEE P1520 Working Group. IEEE P1520: Proposed IEEE standard for application programming interfaces for networks – web site. http://www.ieee-pin.org/.

[7] K. Calvert, Ed. Architectural Framework for Active Networks. Technical report, AN Architecture Working Group, 2000.

[8] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symp. on Operating Systems Principles*, pages 52–65, St. Malo, France, October 1997.

[9] Tal Lavian, Robert Jaegeer, and Jeffrey Hollingsworth. Open programmable architecture for java-enabled network devices. In *Proc. of the Seventh IEEE Workshop on Hot Interconnects*, Stanford University, CA, August 1999.

[10] S. Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *The Fourth IEEE Conference on Open Architectures and Network Programming*, Anchorage, Alaska, April 2001.

[11] Larry Peterson, Yitzchak Gottlieb, Mike Hibler, Patrick Tullmann, Jay Lepreau, Stephen Schwab, Hrishikesh Dandekar, Andrew Purtell, and John Hartman. An OS Interface for Active Routers. *IEEE Journal on Selected Areas in Communications*, (19):473–487, March 2001.

[12] S. Murphy, Ed. Security Architecture Draft. Technical report, AN Security Working Group, 2000.

[13] Oliver Spatscheck and Larry Peterson. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Symp. on Operating System Design and Impl.*, pages 59–72, February 1999.

[14] Carl A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, September 1995.

[15] David Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th Symp. on Operating Systems Principles*, pages 64–79, December 1999.

[16] David Wetherall, John Guttag, and David Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH 98*, San Francisco, CA, April 1998.

[17] Lixia Zhang, Steve Deering, Debra Estrin, Scott Schenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(9):8–18, September 1993.