NodeOS Interface Specification

AN Node OS Working Group

January 10, 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 [8]. 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 [17, 16], PLAN [1, 6], and CANES [4]. 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 [9]. 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 [5]: 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. Said another way, 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 [12].

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 abstractions, and explains the relationships among them.

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 memory pool, and a thread 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 [14]. 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 chosen to implement independent packet flows in their own domains (Domain C through Z), while

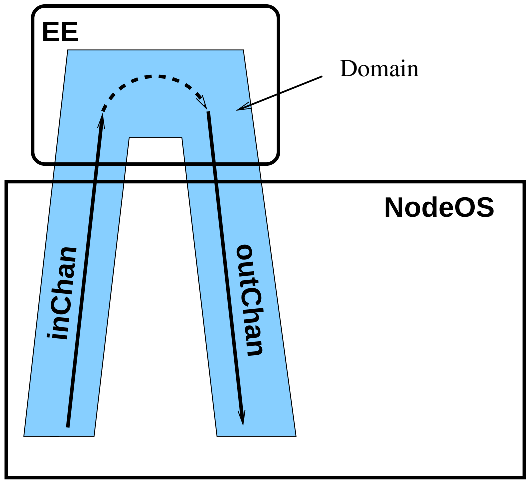

Figure 1: A domain consists of channels, memory, and threads needed for EE-specific processing.

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.)

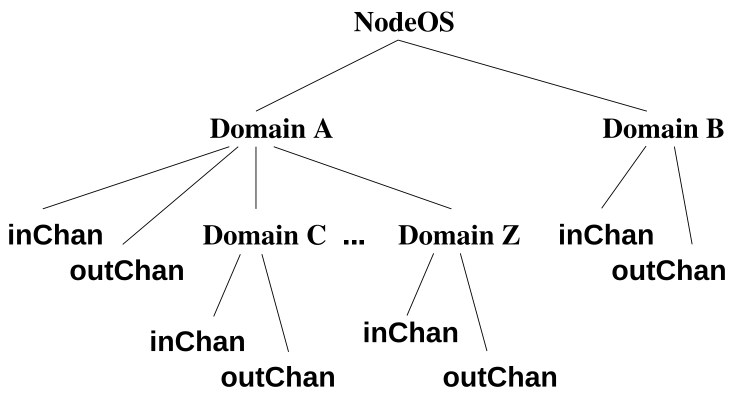

Figure 2: Domain hierarchy.

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 [15]), 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 (identity). 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 exepct some EEs 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 to enforces memory limits at domain granularity. Unfortunately, this approach is too fine-grained for a JVM-based EE in which sharing memory between domains is common. In the model where the NodeOS enforces domain-based limits, destroying a domain would require returning the specific pages of memory that were explicitly allocated to the domain. 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.

Memory pools can be arranged hierarchically to allow constrained sharing between pools. The hierarchy of mempools is not used to control the propagation of resources; rather, it is intended as an access control mechanism. Specifically, a "child" mempool does not inherit its memory resources from its "parent"; those resources come from domains that are attached to the pool. Instead, the mempool hierarchy allows for sharing of memory between pools: a parent may see all of a child's memory while limiting what the child may see of its own. The semantics of the mempool hierarchy are motivated by the desire to support multiple address spaces.

Threads gain access to memory from their associated memory pool using a mmap-style interface. A thread can map an address range against a pool causing all memory resources (e.g., physical memory pages) to be charged to the pool. An "unmap" operation returns resources to the pool while a "protect" operation controls various attributes of mapped memory. Because many EEs will not want to directly manage their address space with mmap-style primitives, it is expected that the NodeOS will also define an allocation library that supports the familiar malloc-style interfaces.

2.4 Channels

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* (cutChan), 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 [18].

Cut-through channels both receive and transmit packets. A cutChan 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 cycles and memory used by a cutChan are charged to the containing domain's thread and memory pool. Figure 3 illustrates an example use of cut-through channels, in which "data" packets might 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 inChans and outChans are composed of modules built into the NodeOS. For example, "ip", "udp", or "anep". Components are separated in the specification string 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 interface. For example "if0/ip/udp/anep" specifies incoming ANEP packets tunneled through IP, while "ip/if" specifies outgoing IP packets. The address specification 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 specification (e.g., describing UDP addresses). cutChan protocol specifications have an identical syntax with the addition of a '|' symbol to denote the transition from incoming packet processing to outgoing packet processing; e.g., example, "ip/udp|udp/ip".

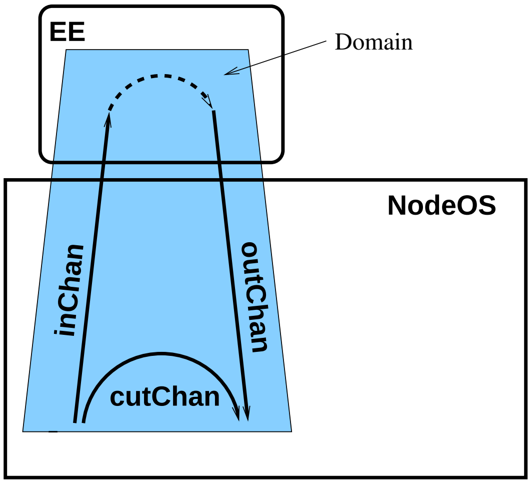

Figure 3: A domain with a cut-through channel.

Simply specifying the protocol and addressing information is insufficient when an EE wants to demultiplex multiple packet flows out of a single protocol (e.g., from a single UDP port). The demux key passed to the inChan specifies a set of (offset, length, value, mask) 4-tuples. These tuples are compared in the obvious way to the "payload" of the protocol. The "payload" is defined as the non-header portion of the packet for whatever protocol specification was given. For example, with a raw "if" specification, the payload is everything after the physical headers; with an "if/ip/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 distinction is in the processing done on the packets by the NodeOS. 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/ip/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 where 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 module names (e.g., "if0/ip/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 API defined for programmable networks [7, 11]. In the latter case, the EE that creates the cutChan is able to *name* the extension (e.g., "if0/ip/extension/if1") and specify parameters according to a standard interface, but 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_ACD datatype, 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. In order to accomodate environments in which EE file sharing is desireable, however, an interface (not yet specified) will be provided to allow EEs to access the entire namespace. This interface will most likely be a function of NodeOS configuration parameters, or perhaps the credentials with which the EE is instantiated.

2.5.2 Concurrency Model

EEs may share information through the use of shared memory regions, which are created with the combination of shm_open and mmap, as described in the next section. A non-persistent file object is first created with shm_open, which allows the specification of a name, as well as access rights and other options. Once the file object is created, EEs may then mmap the object to create a region of memory that is either private (not shared), or shared among the EEs mapping that file object.

The concurrency model by which EEs share this memory has not yet been defined, but one possibility is a versioning scheme based on Lamport's shared clock algorithm[10], as described in [3]. *Version* numbers are employed to allow a single writer to synchronize with multiple readers by indicating that changes are in-progress or complete within a shared memory region. The version numbers are contained within a *blackboard* structure in the first page of each shared region. The central mechanism will be for a single writer to signal to the readers when a change is in progress, and when the change is complete. Readers that detect changes to the data during their reads are expected to retry the read operation.

In order to detect these changes, each blackboard contains a version number V and a size counter S. Each counter is constructed from two unsigned 32 bit integer variables, (V0 and V1, and S0 and

S1, respectively). At the time of creation, these variables are set to 0 by the Node OS. Reads and writes to V0, V1, S0, and S1 are assumed to be atomic.

There are two distinct update operations, *appending* new data to the shared memory region, and *updating* data anywhere within the region. To append bytes to the current version of the shared memory region, the writer increments S0, then stores the data into the shared memory region, and then increments S1. A reader synchronizes with this update by testing V0= =V1, and S0= =S1 before retrieving data from the shared memory region, and then testing that V0 is unchanged after completing the read. Data stored in the first S bytes of the region are valid. (The reader can optionally check that more data has not been appended to the region by checking that S0 is unchanged as well.)

To update data within the region, the version counter is used to indicate that new values may have been stored throughout the shared memory region. To perform an update, the writer first increments V0, then stores new values into the shared memory region, then (optionally) stores new sizes into S0 and S1, and signals completion of the operation by incrementing V1.

3 Interface

This section defines the interface for the five primary abstractions, plus additional miscellaneous interfaces provided by the NodeOS.

3.1 Conventions

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

3.1.1 Symbol Namespaces

Nodeos implementations reserve, at least, the prefixes "an_", "ani_", "nodeos_", and "AN_" 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 CPP macro namespace will fall within these defined prefixes. Note that all of the types and functions defined in this document use the "an_" prefix, while all the CPP macros use the "AN_" prefix.

3.1.2 Specifications vs. Objects

There are two kinds of structures 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 until it is explicitly destroyed. Note that non-structure parameters (such as char * or void *) are not part of this convention.

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 EE must ensure that the specification and any storage referenced by the specification are not mutated (or reclaimed) until the NodeOS API function returns. 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. This restriction also implies that an EE may pass the same specification to multiple API calls simultaneously. 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" as described previously, 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, however, can be reused immediately after the an_domainCreate function returns.

3.2 Domains

an_Domain an_domainCreate(void * mem, an_Credentials c, an_ThreadPoolSpec tp, an_MemSpec vm, an_NotifyFunc termhandler):

Initialize the given block of memory as a new domain using the specified credentials, the specified thread pool and memory resources. If not NULL, this block of memory must the be at least of size AN_SIZEOF_DOMAIN. The form and meaning of the credentials are defined by the Security architecture [13]. The NodeOS is considered the root domain. The NodeOS does a callback to function termhandler in the parent domain if a child domain is being terminated. The signature of an an_NotifyFunc is implementation specific.

an_ThreadPoolSpec and an_MemSpec specify how the thread and memory resources, respectively, are to be initialized.

```
typedef struct an_ThreadPoolSpec {
    u_int32_t     max;
    u_int32_t     cps;
    an_Timespec   yield_limit;
    an_Schedule   s;
    an_InitFunc   f;
} * an_ThreadPoolSpec;
```

Specifies that max threads can run concurrently in the domain, that cps cycles-per-second of normalized CPU bandwidth is to be allocated to those threads, that scheduler s should be used to schedule the domain's threads, and that function f is to be called to initialize the domain. The yield_limit 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_MemSpec {
```

```
    u_int32_t    npages;
} * an_MemSpec;
```

Specifies that the domain is allowed to use npages pages of physical memory. The domain must be associated with a memory pool before the pages can be accessed.

an_Error an_domainDestroy(an_Domain fid): Destroy a domain fid 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_Error an_domainId(an_Domain * retval): Returns the current domain ID.

3.3 Thread Pool

an_Error an_threadSelf(an_Thread * retval): Returns the current thread ID.

an_Error an_threadStart(an_Domain fid, void (*f)(void *arg), void * arg, an_Thread * retval): Start one of the domain's threads at function f with argument arg, and store its thread ID in retval.

an_Error an_threadExit(void): Return currently executing thread to the thread pool.

an_Error an_threadYield(void): Yield the processor from the current thread.

an_Error an_threadSleep(const an_TimeSpec delay): Delay the current thread for at least the given number of seconds and nanoseconds.

an_Error an_threadSetData(void * data): Set a per-thread pointer for the current thread.

an_Error an_threadGetData(void ** retval): 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.

an_Error 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_Cond to 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.

an_Error 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.

an_Error 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, using the region of memory pointed to by mem. If not NULL, the block of memory must have size of at least AN_SIZEOF_MUTEX. The new mutex is initially unlocked.

an_Error an_mutexDestroy(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_mutexLock(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_mutexTryLock(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_mutexUnlock(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 using the memory pointed at by mem. If not NULL, the block of memory must have size of at least AN_SIZEOF_COND.

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_Error an_mempoolCurrent(an_MemPool * retval): Get the mempool associated with the current thread's domain.

an_MemPool an_mempoolCreate(an_MemPool parent, void (*callback)(an_MemPool pool, int curr, int limit)):

Create a new mempool with no associated domains. Use the given block of memory to create the structure. If not NULL, the block must have size of at least AN_SIZEOF_MEMPOOL. The parent parameter defines this pool's position in the pool hierarchy. The callback routine is invoked whenever the NodeOS detects that the indicated pool is over its page limit. Parameters to the callback function include the pool, the current page consumption of the pool (including the amount that puts it over limit), and the page limit of the pool. The callback function should use an_mempoolUnmapRegion to free up memory.

Issue: the current callback parameters may not be sufficient for all environments. For example, a VM environment may need a way to determine how many physical pages are associated with a given VA range.

an_Error an_mempoolDestroy(an_MemPool pool): Destroy a mempool. All domains that were in the pool are implicitly removed (see an_mempoolRemoveDomain) and all child pools are (recursively) destroyed.

Issue: possibly we should just require that the pool be empty before calling destroy, that would avoid all the cascading side-effects. Not sure it is a reasonable thing though, as a parent domain (EE) might not be able to destroy a pool it created if a child created a subpool in it. Since this interface is not exported to AAs, the only situation where this would be a problem is if one EE starts another.

an_Error an_mempoolAddDomain(an_MemPool pool, an_Domain fid): Donates the memory resources associated with the given domain to the pool. The domain must not currently be associated with any other pool. This call increases the limit on the number of physical pages that can be allocated to mappings on this pool. Note that the added domain is not guaranteed to get the memory resources indicated by its own limit; other domains in the same mempool could use those resources if the EE allows them to. All threads in the domain gain access to the memory mapped in the pool.

Issue: perhaps a domain should be associated with a mempool at domain creation time instead of as a separate operation. It isn't clear why you would want a domain that was not assigned to a memory pool, or why you might move a domain from one pool to another.

an_Error an_mempoolRemoveDomain(an_MemPool pool, an_Domain fid): Removes the memory resources associated with the domain from the pool. This call decreases the limit on the number of physical pages that can be allocated to mappings on this pool; it does not directly remove any specific pages. This call may leave the pool over-budget and trigger a reclamation callback. All threads in the domain lose access to memory as a result of this call.

Issue: as with adddomain, we may not want an explicit interface for detaching a domain from a pool. Perhaps it will just happen when a domain is destroyed.

The following calls are used to control accessibility of memory in a pool. Memory can be mapped into or unmapped from the address space implied by a memory pool. The memory resources used to provide these mappings (physical memory and possibly backing store) are charged against that pool. These calls are similar in spirit to the POSIX mmap, munmap and mprotect calls. Protection values include the standard: PROT_READ, PROT_WRITE and PROT_EXEC as well as PROT_WIRED for wiring physical memory. The only flag currently defined is MAP_FIXED to tell mapregion to map memory at the indicated address.

an_Error an_mempoolMapRegion(an_MemPool pool, an_Size len, an_Prot prot, an_Flag flags, an_Addr * addr): Allocate a region of memory from the pool, mapped as indicated. If the flags parameter includes MAP_FIXED, the region must be mapped at *addr or else the call will fail. Otherwise, the NodeOS is free to locate the region whereever it wants, returning the address in *addr. This call might fail or trigger a reclamation callback if physical memory allocation is done eagerly. There may be implementation specific alignment constraints on the address and length parameters and not all implementations may support MAP_FIXED.

an_Error an_mempoolUnmapRegion(an_MemPool pool, an_Addr addr, an_Size len): Deallocate a region of memory. Any physical pages associated with the indicated region are credited to the mempool.

Issue: must all pages of the indicated range be allocated for this call to work or do we just ignore unallocated areas? Must the range passed exactly match a range that was allocated with mapregion? I would say no to both, but I'm not sure we want to mandate a particular behavior.

an_Error an_mempoolProtectRegion(an_MemPool pool, an_Addr addr, an_Size len, an_Prot prot): Respecify the access permissions on a region of memory. Specifying PROT_WIRED may trigger reclamation callbacks or cause the call to fail.

Issue: must all pages of the indicated range be allocated for this call to work or do we just ignore unallocated areas? Must the range passed exactly match a range that was allocated with mapregion? I would say no to both, but I'm not sure we want to mandate a particular behavior.

an_Error an_mempoolPageSize(an_MemPool pool, u_int32_t * retval): Returns the "page" size in bytes for the indicated pool. This call is needed since memory limits are expressed in units of pages, and the page size may determine alignment constraints for mapping.

Issue: this call might be replaced by a system-wide getpagesize call if per-pool page sizes are deemed unnecessary.

an_Error an_mempoolRegionInfo(an_MemPool pool, an_RegionInfo info[], u_int32_t * nelt): Optional. Returns an array of descriptors for all mappings associated with a pool. A descriptor would contain at least the address, size and protection of a region. It might also contain a hint as to how many physical pages are allocated to the range (for use by the revocation callback). On calling, *nelt contains the number of elements in info. On return, *nelt contains the number of entries filled in by the NodeOS.

3.5 Channels

3.5.1 Channel creation functions

an_InChan an_inchanCreate(void * mem, an_Domain fid, an_DemuxKey demuxKey, char * protocolspec, char * addrspec, an_ThreadPoolSpec threadpool, an_BufferPool bufferpool, an_ChanRecvFunc deliverfunc, void * deliverarg):

Creates an input channel in the region of memory mem. If not NULL, this region must have size of at least AN_SIZEOF_INCHAN. Only packets matching the given demuxkey are handed to the deliver function. Processing specified by the protocolspec argument is performed on the packet. The demuxkey does not match against the headers used in the protocol processing. The given buffer pool is used for packet buffer memory. Threads are scheduled out of the owning domain's ThreadPool to run the deliverfunc.

protocolspec must specify, at a minimum, an interface on which to receive packets. See the Protocol Components description below.

addrspec specifies the addresses to match. It is a restricted filter. The format of this specification is dependent upon the protocolSpec. See the Protocol Components description below.

threadpool specifies the CPU resources for the channel.

Issue: a threadpool spec may be too general. It allows maximum flexibility but raises questions about its relationship to the domain threadpool (e.g., is it a subset of the domain pool?) and may just be overkill.

deliverfunc must point to a function. If it is NULL the inchan will not be activated. This is required when creating an inchan to pass to an_cutchanCreate().

deliverarg will be passed verbatim to the deliverfunc. The exact type of an_ChanRecvFunc is left to the NodeOS implementation. It should include the deliverarg, the head of the packet that was delivered, and a pointer to the first byte of the "payload" and perhaps the first byte after that matched by the demuxKey.

Returns NULL if there is an error.

Issue: an_BufferPoolis undefined.

an_OutChan an_outchanCreate(void * mem, an_Domain fid, char * sendprotocol, char * addrspec, an_LinkQOSSpec linkqos):

Create an output channel on which packets can be sent in the region of memory mem. If not NULL, the block of memory must have size of at least AN_SIZEOF_OUTCHAN.

The sendprotocol and addrspec define the processing and headers that are attached to any packets sent on this channel.

Returns NULL if there was an error creating the channel.

an_CutChan an_cutchanSplice(void * mem, an_Domain fid, an_InChan inchan, an_OutChan outchan, char * protocolprocessingspec):

Create a an_CutChan that processes packets from the given inchan and pushes them through the processing specified by protocolprocessingspec and finally out the given outchan. If not NULL, the block of memory mem must have size of at least AN_SIZEOF_CUTCHAN.

Note that the inchan provided must have its an_deliverfunc set to NULL.

It will probably be illegal to send packets on an outchan that is spliced into a cutchan.

The protocolprocessingspec tells the NodeOS what sort of processing is done on the packets. E.g., "ip/udp/magicIn|magicOut/udp/ip". 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.

an_CutChan an_cutchanCreate(void * mem , an_Domain fid, an_DemuxKey demuxkey, char * inprotocolspec, char * outprotocolspec, char * inaddrspec, char * outaddrspec, an_BufferPool bufferpool, an_ThreadPool threadpool):

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.

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 oc, an_Packet p):

Sends packet p on out channel oc. This routine may block.

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 inchan 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 Components

Components that all NodeOSen will support and their definitions and restrictions.

Issue: A compact, flexible and efficient representation of components that does not rely on parsing strings would be appreciated.

“if” : any network interface. Generally, only meaningful when combined with higher-level protocol components since there is no way to specific a meaningful address component when multiple interface types are present.

“if0” : network interface zero (Also, “if1”, “if2”, etc).

“ipv4” IPv4 level processing (fragmentation, reassembly). Must be proceeded by “if/” or “if*N*/”.

“tcp” : TCP processing (ACKs, timeouts, etc). Must be proceeded by “ipv4/”.

“udp” : UDP processing. Must be proceeded by “ipv4/”.

“anep” : ANEP processing (whatever that means). Must be proceeded by “if” 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 InChannels.

UNKNOWN : *Issue: IPv6 modules? Other default Modules?*

3.5.4 Component Address Formats

Address components. For each protocol component, the corresponding address format is provided. The “highest-level” protocol component in a chain determines the address format. *Issue: need to clarify definition of format when provided to inchan, outchan, and cutchan, respectively.*

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

Address is a MAC address in hex. Format is specific to the interface device (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: “0x0000e45907” (the “0x” prefix is optional).

Example: “0x0000e45907:0x800” matches only ETHERTYPE_IP packets.

“ipv4” : “<srcaddr>[:<destaddr>[:<protocol>]]”

<srcaddr> is a dotted quad or hostname. <protocol> is a number between 0 and 256. <destaddr> is a dotted quad or hostname and must be a name for one of the interfaces on the localhost. ’*’ can be used to skip any field.

Example: “*:*:localnet” matches all IPv4 packets destined to the “localnet” interface on the router.

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

Example: “18.34.100.80” matches all IPv4 packets destined to this router from a specific address.

"udp" : "<srcaddr>[:<srcport>[:<destaddr>[:<destport>]]]"

<srcaddr> and <destaddr> are as in the "ipv4" spec address format. <destport> and <srcport> are port identifiers between 0 and 65535.

Example: "*:*:*:80" matches all UDP traffic on port 80 destined to this router.

Example "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>]]]"

?

"anep" : "<typeid>[:<version>]"

<typeid> and <version> are 16-bit and 8-bit integers, respectively, as described in the ANEP RFC.

3.5.5 Demultiplex Keys

Demultiplex keys are used to filter incoming packets. See an_inchanCreate(). 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.

an_DemuxKey an_demuxkeyCreate(void * mem, an_Domain fid):

Create a new demux key. If not NULL, mem must point to a block of memory with at least AN_SIZEOF_DEMUXKEY bytes. It will be initialized.

Returns the new an_DemuxKey object. Returns NULL if there was an error.

an_Error an_demuxkeyDestroy(an_DemuxKey key):

Destroy the given demux key.

an_Error an_demuxkeyMatch(an_DemuxKey key, u_int32_t offset, 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 offset bytes from the beginning of the payload. 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 as long as length.

This function can be called multiple times to create an an_DemuxKey which matches multiple separate sequences. Overlapping sequences will cause an error to be returned and the an_DemuxKeyto be unchanged.

Issue: Sequence and mask may be copied into the demux key (specs) or they may not be (objects). Haven't decided yet. (E.g., semantics of changing sequence *after a call to* an_demuxkeyMatch()*).*

Returns offset + length.

Issue: A way to specify a variable offset would be useful. E.g. to define offset *as the value of the 4-bit quantity starting at bit 4 of the header.*

an_Error an_demuxkeyMatchIPv4(an_DemuxKey key, u_int32_t ipsource, u_int32_t ipdest, u_int8_t protocol):

Match an IPv4 header.

ipsource, ipdest and protocol may be -1, which will match any value. *Issue: Is 0 a better wildcard?*

an_Error an_demuxkeyMatchIPv4UDP(an_DemuxKey key, u_int32_t ipSource, u_int32_t ipDest, u_int16_t srcPort, u_int16_t destPort):

Match an IPv4 UDP header. Note that this match is special in that variable length IP options will be correctly skipped to find the UDP header.

offset should be the beginning of the IP header in the payload.

an_Error an_demuxkeyMatchIPv4TCP(an_DemuxKey key, u_int32_t ipSource, u_int32_t ipDest, u_int16_t srcPort, u_int16_t destPort):

Match a TCP header. Again, like UDP this is a somewhat magical function that skips the variable bits of the IP header.

an_Error an_demuxkeyMatchIPv4UDPANEP(an_DemuxKey key, u_int8_t version, u_int16_t type):

an_Error an_demuxkeyMatchIPv4ANEP(an_DemuxKey key, u_int8_t version, u_int16_t type):

Issues: Other convenience functions are probably useful.

3.6 Filesystem

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

an_Error an_fileClose(an_Domain fid, an_File file): Close the open file referenced by file. See Section 6.3.1.

an_Error an_fileFsync(an_Domain fid, an_File file): Cause all modified data and attributes of file to be moved to permanent storage. See Section 6.6.1.

an_Error an_fileTruncate(an_Domain fid, an_File file, an_Offset length): Cause the file referenced by file to be truncated or extended to length. See Section 5.6.7.

an_Error an_fileLseek(an_Domain fid, an_File file, an_Offset offset, an_Whence whence): seek pointer for the file referenced by file to offset, according to the directive whence. See 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 fid, const char * path, an_ACD acd): The directory given by path is created with the access permissions specified by acd. See Section 5.4.1.

an_Error an_fileRead(an_Domain fid, 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 Section 6.4.1.

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

an_Error an_fileWrite(an_Domain fid, 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 6.4.2.

an_Error an_fileUnlink(an_Domain fid, 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 5.5.1.

an_Error an_fileOpen(an_Domain fid, 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 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 fid, 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 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 fid, 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, fstat behaves identically to the stat function. See Section 5.6.2.

an_Error an_fileMMap(an_Domain fid, an_Size nbytes, an_Prot prot, an_Flag flags, an_File file, an_Offset offset, void ** address): Cause the pages starting at *address to be mapped from the open file referenced by file. The region starts at offset into the file, and extends for nbytes bytes. If the value of *address is non-zero, it is used as a hint to the system, although the actual address may differ if the the flags argument does not include the option AN_MAP_FIXED. The actual address is returned in the location pointed to by address. See Section 12.2.1. The protections for the new region are specified in the prot argument, and is a subset of the following:

- **AN_PROT_EXEC:** Pages may be executed.
- **AN_PROT_READ:** Pages may be read.
- **AN_PROT_WRITE:** Pages may be written.

The flags argument is a subset of the following:

- **AN_MAPPED_SHARED:** Changes are shared.
- **AN_MAPPED_PRIVATE:** Changes are private.
- **AN_MAPPED_FIXED:** Interpret address exactly.

an_Error an_fileMUnmap(an_Domain fid, void * address, an_Size nbytes): Remove any mappings for those entire pages containing any part of the region starting at address and continuing for nbytes bytes. If a mapping to be removed was private, any modifications made in this address range will be discarded. See Section 12.2.2.

an_Error an_fileMSync(an_Domain fid, void * address, an_Size nbytes, an_Flag flags): Writes all modified data to permanent storage, if any. The region starts at address, and extends for nbytes bytes. All whole pages in that region are stored. If the object mapped is non-persistent, or was mapped with AN_MAPPED_PRIVATE, the msync function has no effect. See Section 12.2.4. *Issue: What about flags? Perhaps msync should be synchronous all the time?*

an_Error an_fileShmOpen(an_Domain fid, const char * path, an_Flag flags, an_ACD acd, an_File * retval): Create a non-persistent open file object named by path. This open file object is suitable for use with either mmap or as a generic file that can be read and written. The file object is returned in the location pointed to by retval. See Section 12.3.1. The flags are specified by or'ing an appropriate subset of the following values:

- **AN_O_RDONLY:** open for reading only

AN_O_RDWR: open for reading and writing

AN_O_CREAT: create object if it does not exist

AN_O_TRUNC: truncate size to 0

AN_O_EXCL: error if create and object exists

an_Error an_fileShmUnlink(an_Domain fid, const char * path): Remove the name of the non-persistent open file object named by path. If one or more references to the object exist when the object is unlinked, the name is removed, but the removal of the object contents is postponed until all open and map references to the object have been removed.

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_CANCELLED.

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 Heap

Instead of bothering themselves with the details of memory management, EE's can use a simple heap allocator that supports the following operations.

void *malloc(size_t size): : Return a buffer of the specified size.

free(void *buf): Free the memory associated with the specified buffer.

3.7.3 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 fid, u_int8_t * buf, an_Size len): Create a packet with initial contents contained by buf. The fid 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 fid, an_Packet p, an_Packet * copy): Creates a duplicate of packet p. The fid designates the domain the new packet will be associated with for its lifetime.

an_PacketWalk an_packetwalkCreate(void * mem, an_Domain fid, 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_SIZEOF_PACKETWALK. The fid 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.

3.7.4 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.

3.7.5 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.

3.7.6 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 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`

5 Acknowledgements

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.

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] Glenn Benson. An optimal solution to the secure reader-writer problem. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 251–259, Oakland, California, April 1992.

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

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

[6] 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.

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

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

[9] 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.

[10] L. Lamport. Concurrent Reading and Writing of Clocks. *ACM Transactions on Computer Systems*, November 1990.

[11] 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.

[12] 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*, 2001. To appear.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.