Architectural Framework for Active Networks
Version 0.9

Active Networks Working Group

August 31, 1998

Abstract

This memo defines an architectural model for active networks. It defines the assumptions and goals for the active network and describes the functionality of each node. The approach specifies a set of basic functions from which the end-to-end abstractions presented to the user are implemented.

1 Introduction and Rationale

This document describes an architectural framework for active networks. It lays out the fundamental tenets and objectives, defines the major components and interfaces that make up an active node, and discusses some architectural features common to network services built using such active nodes. The architecture does not specify any particular end-to-end service for the active network. Rather, it identifies the components responsible for implementing such interfaces, and specifies the basic functionality to be provided in support of those components. This document is intended as a "reference model" for the design and implementation of active networks.

The model is quite general. An active network consists of a set of active nodes connected by a variety of network technologies. Each active node runs an operating system and one or more execution environments. The OS is responsible for allocating and scheduling the node's resources (link bandwidth, CPU cycles, and storage). Each execution environment implements a virtual machine that interprets active packets delivered to it. That virtual machine may be completely general—i.e., Turing-complete—or it might simply be a forwarding engine whose "computation" is controlled by the fields of an IPv4 header.

The framework presented here is intended to be consistent not only with current research efforts, but also with possible future production deployments, including commercial networks. The specific roles considered, explicitly or implicitly, in defining this architecture include: node administrations, node vendors, execution environment developers, service designers (that is, third parties who offer prepackaged services built on top of processing environments), and end users.

1.1 Document Organization

This document is organized roughly in three parts. The first part (Sections 1–4) reviews the basic assumptions and objectives and gives an overview of the architecture. The second part (Sections 5–7) describes the function of a generic active node, its major components and interfaces. The third part (Sections 8–9) discusses some global considerations for building practical networks from active nodes.

2 Fundamental Tenets

The following assertions are considered to be "given":

- The *primary* function of the active network is communication and not computation. The network contains some nodes whose sole purpose is to allow sharing of transmission resources. Computation may occur, and indeed computation services could be built upon the active network platform, but the platform itself is not designed to be a general-purpose distributed computing system.
- The unit of multiplexing of the network is the packet (and not, say, circuits).
- Active nodes are interconnected by a variety of packet-forwarding technologies, and this variety will evolve continually. Therefore assumptions about underlying services must be minimized.
- Each active node is controlled by an administration, and no administration controls all active nodes.
- Trust relationships may vary between administrations. In general, trust is explicitly managed.

3 Objectives

The active network provides a platform on which network services can be experimented with, developed, and deployed. That platform represents a "meta-agreement" among the parties concerned with the network (users, service providers, developers, researchers) regarding the fundamental capabilities built into the network, as well as the mechanisms through which they may be accessed and combined to implement new services. Potential benefits of such a capability include faster deployment of new services, the ability to customize services for different applications, and ability to experiment with new network architectures and services on a running network.

The general role of the architecture is to describe the form of the active network platform, its major components and interfaces. As usual, the intent is to be specific but avoid ruling out possible solutions by overconstraining things. Thus, the architecture defines functional "slots" in the active network platform, which may be filled in various ways.

This architecture has been developed with the following objectives in view:

- Minimize the amount of global agreement required, and support dynamic modification of aspects of the network that do not require global agreement.
- Support, or at least do not hinder, fast-path processing optimizations in nodes. In particular, it should be possible to construct an active node that can route plain old IP datagrams at speeds roughly comparable to those of "passive" IP routers.
- Support deployment of a base platform that permits on-the-fly experimentation. Backward compatibility, or at least the ability to fit existing network nodes into the architectural framework, is desirable.
- Scale to very large global active networks.

- Provide mechanisms to ensure the security and robustness of active active nodes individually as well as of the active network as a whole. Robustness mechanisms should be independent of security, so that the consequences of the actions of even the most-authorized user are limited in scope.
- Support network management at all levels.
- Provide mechanisms to support different levels/qualities/classes of service.

4 Architecture

All nodes in the active network provide a common base functionality, which is defined by the *node* architecture. The node architecture deals with how packets are processed and how local resources are managed. A *network* architecture deals with global matters like addressing, end-to-end allocation of resources, and so on. As such, it is tied to the interface presented by the network to the end user. The node architecture presented here is explicitly designed to allow for more than one such "network API".

Several factors motivate this approach. First, multiple APIs already exist, and given the early state of our understanding it seems desirable to let them "compete" side-by-side. Second, it supports the goal of fast-path processing for those packets that want "plain old forwarding service". Third, this approach provides a built-in evolutionary path, not only for new and existing APIs, but also for backward-compatibility: IPv4 or IPv6 functionality can be regarded as simply another API.

4.1 Active Node Components

The functionality of the active network node is divided between the Execution Environments (EEs or just "environments") and the Node Operating System (NodeOS). The general organization of these components is shown in Figure 1. Each EE exports an API or virtual machine that users can program or control by directing packets to it. Thus, an EE acts like the "shell" program in a general-purpose computing system, providing an interface through which end-to-end network services can be accessed. The architecture allows for multiple EEs to be present on a single active node. However, development and deployment of an EE is considered a nontrivial exercise, and it is expected that the number of different EEs supported at any one time will be relatively small.

The NodeOS provides the basic functions on which execution environments build the abstractions presented to the user. As such, it manages the resources of the active node and mediates the demand for those resources, including transmission, computing, and storage. The NodeOS thus isolates EEs from the details of resource management and from the effects of the behavior of other EEs. The EEs in turn hide from the NodeOS most of the details of interaction with the end user.

To provide adequate support for quality of service, the NodeOS allows for allocation of resources at a finer level of granularity than just EEs. When an EE requests a service from the NodeOS, the request is accompanied by an identifier (and possibly a credential) for the principal in whose behalf the request is made. This principal may be the EE itself, or another party—a user—in whose behalf the EE is acting. The NodeOS presents this information to an enforcement engine, which verifies its authenticity and checks that the node's security policy database (see Figure 1) authorizes the principal to receive the requested service or perform the requested operation. A companion document [SecArch] describes the active network security architecture; see also Section 6.

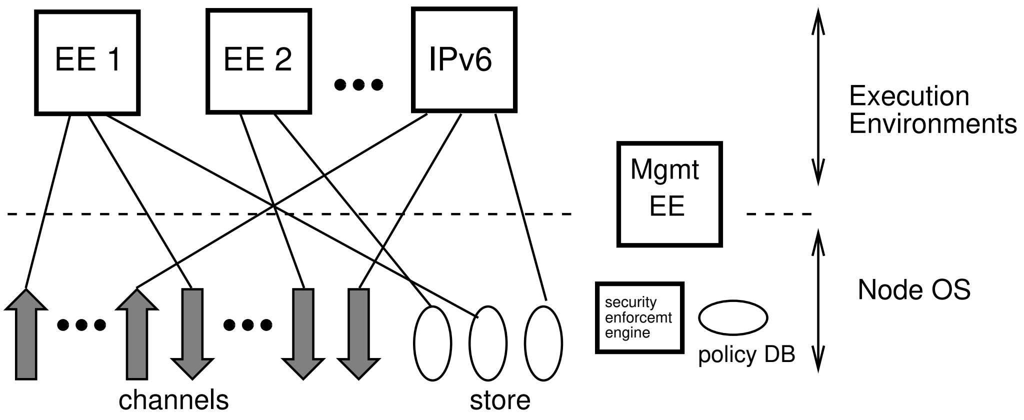

Figure 1: Components

Each node has a distinguished *management execution environment*, through which certain aspects of the node configuration may be controlled. For example, the management EE may allow modification of the security policy database of the node. The management EE may be implemented as part of the NodeOS, or it may be a separate EE that is granted access to special operations by the NodeOS.

In the interest of efficiency, it may be advantageous to have the NodeOS support certain common objects and services such as routing. The motivation for this is discussed further in Section 8. The use of such services, if implemented, is not required in any way by the architecture, and EEs may use them or choose to implement their own.

4.2 General Packet Processing

Execution environments send and receive packets over communication channels. The NodeOS implements these channels using a variety of technologies, including both raw network links (e.g., Ethernet, ATM) and higher level protocols (e.g., TCP, UDP, IP).

The logical flow of packets through an active node is shown in Figure 2. When an active node receives a packet off a physical link, it classifies it based on information in the packet (i.e. headers); packets are placed in channels according to this classification. The classification of incoming packets is controlled by a pattern specified by the EE when it creates the channel (Section 6.3). In the typical case, an EE requests creation of a channel for packets with certain attributes, for example a certain Ethernet type or combination of IP protocol and TCP port numbers. Figure 2 depicts EE 1 receiving ANEP packets of a particular type, encapsulated in UDP datagrams sent to a particular port. EE 2 also receives UDP datagrams containing ANEP packets (presumably with a different ANEP packet type), as well as plain UDP datagrams, ANEP packets in IP datagrams, and IP datagrams matching some pattern (e.g. a specific protocol number, or a particular source and destination pair). It is expected that each incoming packet will match at most one channel. It is the responsibility of the security engine to ensure that a given principal is allowed to create a channel with a particular pattern. Incoming packets that match no description are dropped; local node policy may specify a default (e.g. forwarding) behavior for packets that match no other pattern. Figure 2 shows an example of this as an IP "cutthrough" channel that does not go through

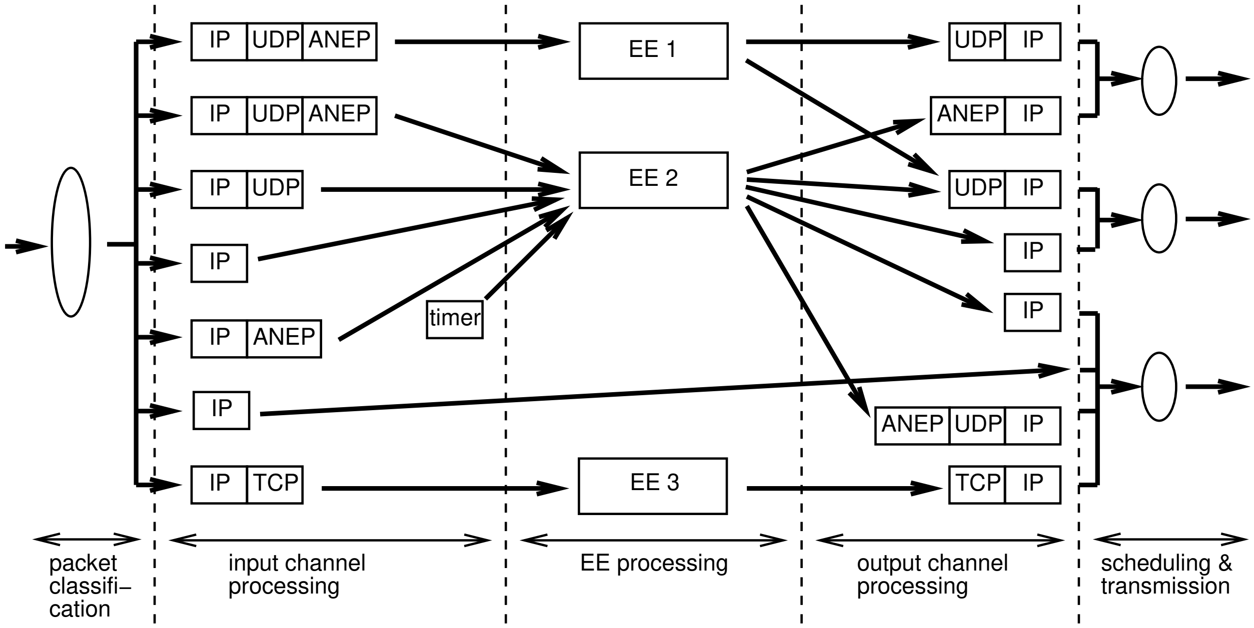

Figure 2: Packet flow through an active node. (Link-level protocols are omitted to save space.)

any EE (see Section 6.3).

To provide for quality of service, the NodeOS has scheduling mechanisms that control access to the computation and transmission resources of the node. These mechanisms isolate user traffic to some degree from the effects of other users' traffic, so that each appears to have its own virtual machine and/or virtual link. When channels are created, the requesting EE specifies the desired treatment by the scheduler(s). Such treatment may include:

- Reservation of a specific amount of bandwidth (transmission or processing), and isolation from other traffic.
- Isolation from other channels' traffic and guarantee of a "fair share" of available bandwidth, the available bandwidth is divided more or less equally among all channels in this class.
- Best-effort, i.e. no isolation from other traffic in this class.

Requests for channels with the first two types of service may be denied by the NodeOS when resources are in high demand. Note that input channels are scheduled only with respect to computation, while output channels are scheduled with respect to both computation and transmission.

4.3 Active Network Encapsulation Protocol

Some means is required to allow users to control the routing of packets to a particular EE. The Active Network Encapsulation Protocol (ANEP) provides this capability. The ANEP header includes a "Type Identifier" field; well-known Type IDs are assigned to specific execution environments. (Presently this assignment is handled by the Active Network Assigned Number Authority.) If a particular environment is present at a node, packets containing a valid ANEP header with the appropriate Type ID (encapsulated in a supported protocol stack) will be routed to channels connected to the indicated EE; these channels are created when the node starts up.

A packet need not contain an ANEP header for it to be processed by an EE. EEs may also process "legacy" traffic—originated by end systems that are not active-aware—simply by setting up the appropriate channels. One example is an EE that provides IPv4 forwarding behavior. Another is an EE that provides enhanced TCP services, implemented by transparent header diddling (EE 3 is shown providing such a service in Figure 2).

The Active Network Encapsulation Protocol also provides a vehicle for other communications with the NodeOS, including:

- Error-handling instructions. When a packet fails to reach the desired EE at a node (perhaps because the node does not support it, or resources were not available), the ANEP header lets the user instruct the Node OS as to the expected response, for example: drop the packet silently, try to forward it, or send an error message. The latter two require an address in the ANEP header that is recognized by the node OS itself. This implies that this mechanism requires at least one address format that can be assumed to be routable by every node.

- Security vouchers. It is impractical for every node in the network to retrieve or store the information (e.g. public key certificate) required to authenticate every packet passing through it. However, it may be practical for a packet to be authenticated just once (based on its originator) when it enters the network, and thereafter be vouched-for node-to-node using keys shared between neighboring nodes. The same technique could be used at inter-domain exchange points. The ANEP header is a natural location for a node-to-node credential.

- Fragmentation and reassembly. A default mechanism is needed for channels that have an MTU mismatch (i.e. a protocol module that has a smaller maximum transmission unit than some higher-level protocol) and include no other protocol providing this service.

4.4 End Systems and Intermediate Systems

This architecture does not distinguish between end systems and intermediate systems. The systems that act as sources and sinks for user packets have the same general structure as do those systems that exist for the sole purpose of sharing transmission resources. Naturally the two types of systems will have different security and other policies. Depending on their size and sophistication, a rudimentary NodeOS may suffice for some end systems. An example would be a handheld mobile unit designed to work with a particular EE.

5 Execution Environments

An execution environment defines a virtual machine and "prograimming interface" that is available to users of the active network. Users control that virtual machine by sending the appropriate coded instructions to the EE in packets. Execution of these instructions will in general cause the state of the EE and the active node to be updated, and may cause the EE to transmit one or more packets, either immediately or at some later time.

The function of the virtual machine provided by any EE is not defined by the architecture. The NodeOS provides a basic set of capabilities that can be used by EEs to implement a virtual machine. Thus, most aspects of the *network* architecture are defined by the EE.

EEs allow users some control over the way their packets are processed. If an EE implements a universal virtual machine —that is, one that can be programmed to simulate other machines— a user may provide a complete description of required packet processing.[1] Other EEs may supply a more restricted form of programming, in which the user supplies a set of parameters and/or simple policy specifications. In any case, the "program" may be carried in-band with the packet itself, or installed out-of-band. Out-of-band programming may happen in advance of packet transmission or on-demand, upon packet arrival; it may occur automatically (e.g. when instructions carried in a packet invoke a method not present at the node, but known to exist elsewhere) or under explicit user control.

5.1 Inter-EE Communication

Execution environments are completely independent of each other; the architecture makes no special provision for EEs to communicate or cooperate with each other. However, inter-EE communication may be possible using those mechanisms.

5.2 Defining and Installing New EEs

An EE provides some end-to-end semantics to network users. To be useful, this semantics must be sufficiently well-defined and specified to enable independent implementation of applications that make use of its services. In addition, deployment of a new EE is a significant undertaking, which may place network resources at risk if the EE misbehaves.[2] In view of these considerations, an EE to be deployed must be described by publicly-available documents that specify the following:

- The Application Programming Interface to the EE. This defines how applications (in end systems) access the services of the EE, including the wire encoding of control information interpreted by the EE. This description may refer to or incorporate other API definitions (e.g., the Java Virtual Machine, or the sockets interface).
- The well-known ANEP packet types for which the EE "listens".
- For EEs that allow users to reserve resources, a method of determining computational and storage requirements for packets processed by the EE (Section 8.2.1).

The architecture is neutral on the existence of "standard" methods for installing new EEs in an active node. If supported, any method of dynamically downloading and installing a new EE should be accessible only to node administrations via the management EE.

6 Node Operating System

The NodeOS is a layer operating between the EEs and the underlying physical resources (transmission bandwidth, processor cycles, and storage). Its existence arises from both the need to simultaneously support multiple EEs and to provide a base level of functionality that is common to every active node.

[1] It is expected that most end users will not program the network, any more than most people program their PCs. Most will select from ready-made or default services provided directly by the EE or by third-party "value-added" service builders.

[2] Section 8.1 discusses possible strategies for reducing this risk

This common functionality includes implementing the underlying channels over which EEs receive and transmit packets, mediating access to node resources such as transmission and compute bandwidth by multiple environments, and supporting common services, such as routing.

6.1 Trust Assumptions

Different trust relationships may exist between the NodeOS and the EEs. In one direction, the EEs must trust the NodeOS; EEs cannot override the NodeOS's policies, but they can augment them with their own policies. In the other direction, the NodeOS may trust some EEs but not others. In those cases where the NodeOS trusts an EE, performance improvements may be possible.

6.2 Principals

The primary abstraction for accounting and security purposes is the *principal*. All resource allocation and security decisions are made on a per-principal basis. In other words, a principal is admitted to an active node once it has authenticated itself to the node, and it is allowed to request and use resources as described below.

There are three primary resource abstractions: *threads*, *memory*, and *channels*. Threads and memory behave exactly as on a traditional OS, although a definition that is specfic to active nodes may be defined at a future time. Channels are defined in the next section.

6.3 Channels

Principals create channels over which they can send, receive, and forward packets. Some channels are *anchored* in an execution environment; principals use anchored channels to send packets between the execution environment and the underlying communication substrate. Other channels are *cutthrough*, 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. Channels are in general full-duplex, although a given principal might only send or receive packets on a particular channel.

It is helpful to define channels in terms of a more primitive building block—*protocol modules*—which implement some well-defined primitive communication service. For example, ANEP, TCP, UDP, IP, an ethernet driver (ETH), and an ATM driver (ATM) are protocol modules with the obvious meaning. Protocol modules can be combined (stacked) to build more complex communication services. Note that a given NodeOS need not implement channels as stacks of protocol modules, although a NodeOS can leverage this implementation to support extensible channels.

Figure 3 distinguishes between anchored and cutthrough channels on an active node connected to a pair of network links (ATM and ETH). In (a), the principal has created a flow through the active node consisting of a pair of anchored channels (depicted as solid lines labeled /ANEP/UDP/IP/ETH and /ANEP/UDP/IP/ATM), and some computation that takes place in the context of the EE (depicted as a dotted line connecting the two channels). Note that the principal need not have created ANEP channels; it could have created channels to send and receive raw ethernet/ATM packets.

In (b), the principal has created a flow through the active node consisting of the same channels as in (a), plus a cutthrough channel that directly connects the ethernet link to the ATM link. This particular example includes IP behavior (e.g. decrementing TTL) on the cutthrough channel. Packets flow over the cutthrough channel without being intercepted and processed by the EE. In this example, the anchored channels (plus the in-EE processing) might by used by an active

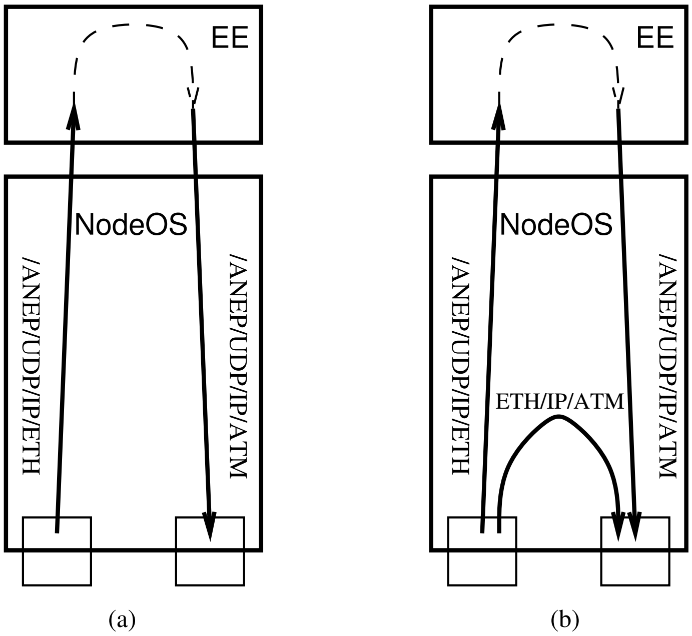

Figure 3: Anchored and Cutthrough Channels

control protocol (e.g., an active version of RSVP) to exchange control messages with its peers. These control messages might then cause the cutthrough channel to be created, with data messages forwarded over this data/cutthrough channel.

Principals create channels by specifying a set of *attributes*. These attributes include the modules that define the behavior of the channel, as well as other properties such as addresses, direction of packet flow, MTU, and quality of service requirements.

In addition to specifying a set of channel attributes, the EE must also provide the NodeOS with enough information to allow the NodeOS to classify incoming packets and demultiplex them to the appropriate input or cutthrough channel. Packet classification is not an issue for output-only channels. We envision this classification information being of one of two forms. Untrusted EEs will specify a classification pattern that will then be interpreted by the NodeOS. Trusted EEs may be allowed to contribute a code fragment that the NodeOS then runs to demultiplex packets to that channel. It is the NodeOS's responsibility to classify each incoming packet according to the information provided by the EEs. Any packet that does not match existing channel is dropped by the NodeOS; it is expected that some or all nodes will support a default channel implementing IP (or IP-like) forwarding semantics.

The final responsibility of the NodeOS is to schedule channels for execution. Scheduling decisions are influenced by both the computational requirements of the channel—which depends on the protocol stack that implements the channel—as well as the bandwidth requirements corresponding to the channel's quality of service guarantee. For example, in Figure 2, two different channels have been created with IP processing for one output interface. One is the standard "best-effort" IP channel, while the other has an associated bandwidth reservation, and is only accessible to EE 2.

7 Interfaces

The major interfaces of the architecture are those between the EEs and the NodeOS, and between EEs and applications. The former is of interest to node vendors and EE developers; the latter is of interest to application developers and EE developers.

7.1 Execution Environment–Node OS Interface

The interface between the Node OS and the EEs does not affect the bits on the wire, and is not directly visible to end user. Rather, it is local to the active node, and may be implemented in node-specific ways, like the system-call interface of an operating system. The specification of this interface consists of "system calls" (methods) provided by the Node OS to the environments. The NodeOS interface is currently under development, and will be defined in a companion document.

7.2 Environment–Application Interface

The interface between the EE and the end-user application is by definition EE-specific. However, it must be precisely specified in order for an EE to be deployed. Some environments may offer services that are accessible via a generic, language-independent interface (a la sockets); others may require that application and network code be integrated, or even written in a particular language (e.g. Java). The intent is that this interface should be more or less independent of the underlying active network infrastructure.

7.3 Management EE Interface

Each active node should support a special environment providing a maintenance interface for the Node OS and the common abstractions it supports. Such an interface would provide control over node policies and parameters, as well as maintenance of the common abstractions (e.g. default routing, see Section 8.3) that are supported by the Node OS and made available to every environment.

8 Network Architecture Considerations

Many aspects of the global active network service seen by a user —the programming model, how packets are forwarded, how state is installed, how resources are reserved, etc. etc.— are determined by the execution environment, and are thus beyond the scope of this document. This section deals with aspects of the network architecture that can or must be directly affected by the node architecture (in particular, the NodeOS). These aspects fall generally into the following categories:

- resource conservation mechanisms;
- information that must be interpreted by both user and node OS; and
- mechanisms that can be shared across EEs for global efficiency.

8.1 Resource Conservation Mechanisms

The active network requires mechanisms to ensure that the network is robust, and to ensure that resource consumption by EEs and users is controlled. In particular, it must not be possible for a malfunctioning or malicious EE or user to prevent other users from receiving adequate resources. Moreover, resource waste (e.g. due to routing loops) should be minimized. EEs will play a role in implementing any such mechanisms, but the problem is complex, and it is possible that different end-to-end mechanisms will be appropriate for different EEs. For this reason it is desirable to delineate the division of responsibility between the NodeOS and the EEs with respect to limiting resource consumption.

The primary resources of interest in the active network are transmission bandwidth, computing bandwidth, and storage. The first of these is arguably paramount—it is the reason for existence of the network. However, in the active network, transmission bandwidth is available only via computation, and so the latter must also be protected. Storage is important both for transmission (buffering) and computation state. As the arbiter of access to individual node resources, the NodeOS must provide allocation and scheduling mechanisms to prevent users/EEs from depriving other users/EEs of access to those resources. These mechanisms must be fine-grained enough that an EE can rely on the NodeOS to protect individual users (of the same EE) from each other, and need not implement such mechanisms itself as another layer on top of the inter-EE isolation mechanisms. Thus, the NodeOS provides each EE (and potentially each user) with a "controlled-load" type of service: a virtual CPU/channel/memory that appears to be unloaded almost all of the time.

The active network will provide for reservation of resources, and waste can occur if reserved resources devoted exclusively to the reserving principal, i.e. are not available to others even when they are not being used by that principal. In order to preclude this possibility from the outset, the following fundamental requirement is imposed:

Resource Conservation Rule 0: *All scheduling algorithms are work-conserving.*

Resources are also "wasted" when they are consumed by useless transmissions or computations. Obvious sources of these include malfunctioning EEs or user programs; however, they may also arise from naturally-occurring but transient conditions such as routing loops, network congestion, etc. It is desirable to bound effects of these and other failure modes.

The classic example of a mechanism to bound is the IP TTL field, which bounds the number of times any individual packet will be transmitted in an IP network. The mechanism states that the TTL of a forwarded packet is one less than the TTL of the received packet. This mechanism requires state, but it is carried in the packet, and need not persist at any node. Unfortunately, in an active network the relationship between the packets received and transmitted by a node is at best unclear and at worst arbitrary. Therefore, the implementation of such mechanisms must be left to the individual EEs.

8.2 EE-Independent Information

Where the NodeOS must deal directly with information provided by or related to the end user, common syntax and semantics are required so the NodeOS can interpret the information independent of the particular EE invoked by the user. One example of such information is the quantification of resources for reservation purposes. Security credentials are another example.

8.2.1 Quantifying Resource Needs

As discussed above, information about resource demands and usage needs to pass between the user and the active network. This information must quantify transmission, computation, and/or storage.

The bit is universal as the quantitative measure of information transmission. Similarly, the byte is uniformly understood as a measure of storage capacity. For our purposes, it is probably a good idea to add a time dimension, to provide a bound on the effect of storing a byte.

How to quantify computation requirements is less clear. Different active nodes may have different underlying hardware architectures and different node operating systems, which impose different overheads. Moreover, the "program" whose computation is being described is EE-dependent, yet the quantity must be expressed in EE-independent terms. The computation involved in input and output channel processing must also be accounted for; it is not clear that the user will even be aware of the existence of such processing, much less how much it varies from node to node.

One possibility is to use "RISC cycles" as the unit of computation, and to require that user-visible quantities always be given relative to the EE-specific code. (This might be user-supplied code, or it might be "standard" functionality built into the EE.) RISC cycles are essentially atomic, and it should be possible to define conversions for other kinds of hardware. Each active node would be responsible for translating "standard RISC cycles" to/from its own computing resource measurements, and accounting for all protocol processing on the input/output channels.

For such a metric to be useful, the correspondence between the "source code" and the "object metric" of RISC cycles must be clear, or at least well-defined. In the case of standard code blocks, the code vendor would also supply the cycle requirements for specific classes of inputs to the code. For user-supplied code, the user might have to estimate, perhaps by referring to standard "benchmark" programs, whose requirements are published; alternatively, tools might be provided to automate the estimation based upon generic formulas.

For all three resources —but especially for memory and computation— precision is likely to be limited. Moreover, the range of values to be specified is likely to be extensive. Therefore a logarithmic metric is recommended for measures of memory and computation.

8.2.2 Security Credentials

The node OS must pass security information (e.g. public-key certificates) to components of the security architecture [SecArch]. A standard syntax and semantics for such information is needed.

8.3 Common Objects and Services

The Node OS may, in the interest of efficiency, support certain common abstractions that can be used by any environment. An important example of such an abstraction is a routing table, which maps addresses to next hops. Rather than requiring each environment to define and maintain its own routing table structure, the Node OS can support a generic routing table object. It should be possible for an environment to create its own instances of routing tables and update and access them. Whether such an instance would be accessible to other environments should be under the control of the creating environment.

The advantage of having the NodeOS implement such facilities is that greater global efficiency may be achieved by having common underlying implementations. For example, a NodeOS-supported routing table abstraction might provide an efficient interface to a high-performance forwarding engine. The disadvantage of using a common routing table abstraction is that it may impose restrictions, for example an assumption that a "longest-prefix" match is used. For that reason, environments can always choose to implement their own abstractions on top of more primitive facilities provided by the NodeOS. (Note that "routing table" in the above paragraph refers to the abstraction itself, and not to the information in the abstraction. It may in fact be useful for a NodeOS also to implement routing protocols and keep a specific globally-accessible routing table updated; however, that is a separate consideration.)

An important common service is a "loading" mechanism that handles the retrieval and installation of user-specific code in the node's state store. How to implement such a mechanism in an environment-independent way is a challenge. More research and experience are needed. One possibility is to allow an environment to specify a "thunk" to be executed whenever it incurs a "miss" on the (common) object store. For example, a miss generated by a packet could be forwarded to the node that last processed the packet, as is done in [ANTS]. Another possibility is to allow the user code to handle the loading explicitly [SANE].

The number and nature of common abstractions to be supported is an issue for further research.

9 Evolution Strategy

As experience is gained with active network applications and infrastructure, it will become clear which features are generally useful. Features useful to all environments should be supported by the node OS. Functions may also migrate into the Node OS for reasons of performance, as discussed in the previous section.

Changes to EEs' APIs virtual machines should occur slowly, if at all. Major changes to the platform should be handled by defining a new environment. The architecture allows other aspects of node functionality (e.g., the Node OS–environment interface) to evolve without affecting the programming platform visible to users.

10 Status and Acknowledgements

This document is the result of discussions and contributions from the DARPA active networks research community. It represents a snapshot of current understanding, and should be expected to continue to evolve.

A number of people have contributed to this and earlier versions of this draft. Larry Peterson contributed much of Section 6, and also suggested RISC cycles as the unit of computation. Bob Braden made a number of useful suggestions. Helpful inputs on this version were also provided by Bobby Bhattacharjee and Sandy Murphy. This and earlier versions owe much to a number of people including those folks and John Guttag, Gary Minden, David Wetherall, Ellen Zegura, and the authors of various other documents including [TW96, ANTS, Braden, SANE, SwitchWare].

Comments are solicited, and should be sent to the mailing list:

`ActiveNets_Wire@ittc.ukans.edu`

or to the editor:

Ken Calvert
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046
Phone: +1.606.257.6745
Mail: calvert@dcs.uky.edu

References

[ANEP] D. Alexander et al, Active Network Encapsulation Protocol. Draft, July 1997. Available at http://www.cis.upenn.edu/ switchware/ANEP/.

[SANE] D. Alexander et al, A Secure Active Network Environment Architecture. http://www.cis.upenn.edu/ angelos/sane.ps.gz

[Braden] B. Braden, Requirements for Node Environments in support of the USC/ISI ARP Project. Active Nets Workshop, March 1998.

[SwitchWare] C. Gunter, S. Nettles and J. Smith, The SwitchWare Active Network Architecture. Active Nets Workshop, March 1998.

[SecArch] S. Murphy, ed. Security Architecture Draft. AN Working Group Draft.

[ANTS] D. Wetherall, J. Guttag, and D. Tennenhouse, ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, IEEE OPENARCH'98, San Francisco, CA, April 1998. See also http://www.sds.lcs.mit.edu/activeware/ants/.

[TW96] D. Tennenhouse and D. Wetherall, Towards an Active Network Architecture. Computer Communication Review, 26(2), April 1996.