Active Network Working Group
Request for Comments: ??
DRAFT

K. L. Calvert, ed.
University of Kentucky
July 27, 1999

Architectural Framework for Active Networks
Version 1.0

Contents

1 Introduction and Rationale

This document describes an architectural framework for active networks. It lays out the assumptions 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. Because one of its objectives is to support a variety of service models, the architecture itself does not specify any particular end-to-end service for the active network. Rather, it identifies the components responsible for implementing such services, and describes, in a general way, the basic functionality to be provided in support of those components. The detailed specification of the basic node services is presented in a separate document, which is expected to continue to evolve. It is expected that particular network architectures and end-to-end services will be defined separately (for example, see [PLAN, ANTS]). The main purpose of this document is to serve 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 nodes (not all of which need be "active") connected by a variety of network technologies. Each active node runs a Node Operating System and one or more Execution Environments. The NodeOS is responsible for allocating and scheduling the node's resources (link bandwidth, CPU cycles, and storage). Each Execution Environment (EE) implements a virtual machine that interprets active packets that arrive at the node. Different EEs define different virtual machines; one might be completely general (Turing complete) while another's computation simply forwards packets under control of the fields of an IPv4 header. Users obtain services from the active network via Active Applications (AAs), which program the virtual machine provided by an EE to provide an end-to-end service.

This approach has several desirable characteristics: it allows for ongoing experimentation with different network service models; it supports for fast-path processing of non-active network traffic; and it provides for evolution of services in a backward-compatible manner.

This document is intended to be consistent not only with ongoing 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, active application designers (that is, third parties who offer prepackaged services built using the programming interfaces provided by specific execution 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 of the architecture and gives an overview of the framework. The second part (Sections 4–8) describes the components and major interfaces, and considers implications of the architecture for implementations on a high-performance switch or router. The third part (Sections 9–10) discusses some considerations for building practical networks from active nodes. Companion documents describe the functional interface provided by the NodeOS [NodeOS], and the security architecture for the active network [SecArch].

1.2 Status

This document reflects the "rough consensus" of the DARPA active networks research community, as expressed in workshops at Tucson (March 1998), Atlanta (July 1998), New York (November 1998), and in discussions on the "ActiveNets Wire" mailing list. The current state of our understanding and experience is such that for many problems the best architectural solutions have yet to emerge; in such areas this document attempts to highlight the key issues.

2 Assumptions

The following are considered to be "given":

- The unit of multiplexing of the network is the packet (and not, say, the circuit).
- The *primary* function of the active network is communication and not computation. The network contains some nodes whose primary reason for existence is to switch packets and thus 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.
- Active nodes are interconnected by a variety of packet-forwarding technologies, and this variety will evolve continually. Therefore assumptions about underlying technologies must be minimized.
- Each active node is controlled by an administration, and no single administration controls all active nodes.
- Trust relationships between administrations will vary. Trust needs to be 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 ("standardization") 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. The *architecture* should not preclude active nodes from performing standard IPv4/IPv6 forwarding at speeds comparable to non-active 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. This requirement applies primarily to the network architecture(s) realized using the node platform described here. The main implication for the node architecture is a requirement that network-scale parameters (e.g. number of principals using the entire active network) not be exposed at the individual node level.
- Provide mechanisms to ensure the security and robustness of active nodes individually. As with scalability, global security and robustness is the responsibility of each individual network architecture. However, the stability of individual nodes is necessary for that of the entire network.
- Support network management at all levels.
- Provide mechanisms to support different levels/qualities/classes of service.

4 Overview of the Architecture

Active nodes provide a common base functionality, which is described in part by the *node* architecture. The node architecture deals with how packets are processed and how local resources are managed. In contrast, 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 programming interface presented by the network to the applications using it. The node architecture presented here is explicitly designed to allow for more than one "network API" and network architecture.

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", via an EE that provides this 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 EE.

4.1 Major Components

The functionality of the active network node is divided among the Node Operating System (NodeOS), the Execution Environments (EEs), and the Active Applications (AAs). The general organization of these components is shown in Figure 1. Each EE exports a programming interface or virtual machine that can be programmed or controlled 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 small.

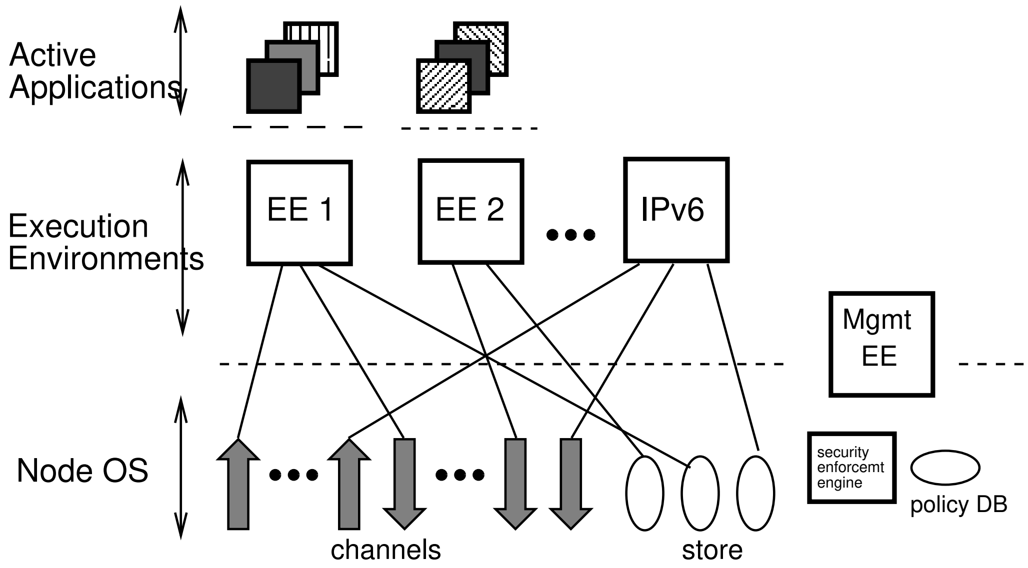

Figure 1: Components

The NodeOS provides the basic functionality from which execution environments build the abstractions presented to the active applications. As such, the NodeOS manages the resources of the active node and mediates the demand for those resources, which include 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—an end user—in whose behalf the EE is executing an active application. 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.

Each node has a distinguished *management execution environment*, through which certain aspects of the local node configuration and policy may be controlled. Examples of the system management and control functions that may be performed via the management EE include:

1. Maintain the security policy database of the node.
2. Load new EEs, or update and configure existing EEs.
3. Support the instantiation of network management services (of the types specified in [**?**]) from remote locations.

Any management functions that can be initiated or controlled via packets directed to the management EE, must be cryptographically secured and subject to the policies specified in the security policy database.

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 general flow of packets through an active node is shown in Figure 2. When a packet is received on a physical link, it is first classified based on information in the packet (i.e. headers); this classification determines the input channel —including protocol processing— to which the packet is directed. Classification of incoming packets is controlled by patterns specified by the EEs. In the typical case, an EE requests creation of a channel for packets with certain attributes, such as a given Ethernet type or combination of IP protocol and TCP port numbers; this request may be issued for the EE itself, or on behalf of an active application. After input channel processing the packet is handed off to the EE that requested creation of the channel; note that this may involve the packet waiting in a queue. Figure 2 depicts EE 1 receiving ANEP packets (see below) encapsulated in UDP datagrams, e.g. those with a particular destination port number. EE 2 also receives UDP datagrams containing ANEP packets (presumably with a different port number and/or 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 the responsibility of the NodeOS and security engine to ensure that the requesting principal is permitted access to the set of packets implied by the pattern associated with a channel creation request. Incoming packets that match no pattern are dropped.

On the output side, EEs transmit packets by submitting them to output channels, which include protocol processing as well as output scheduling. Thus, the general progression is link input, classification, input protocol processing, EE/AA processing, output protocol processing, scheduling, and link transmission. Note that in general it need not be the case that each transmitted packet corresponds to some received packet; EEs may aggregate packets for periodic forwarding, or even generate them spontaneously.

The NodeOS also provides more sophisticated levels of access to both the computation and the transmission resources of the node. These scheduling mechanisms isolate different classes of traffic from the effects of

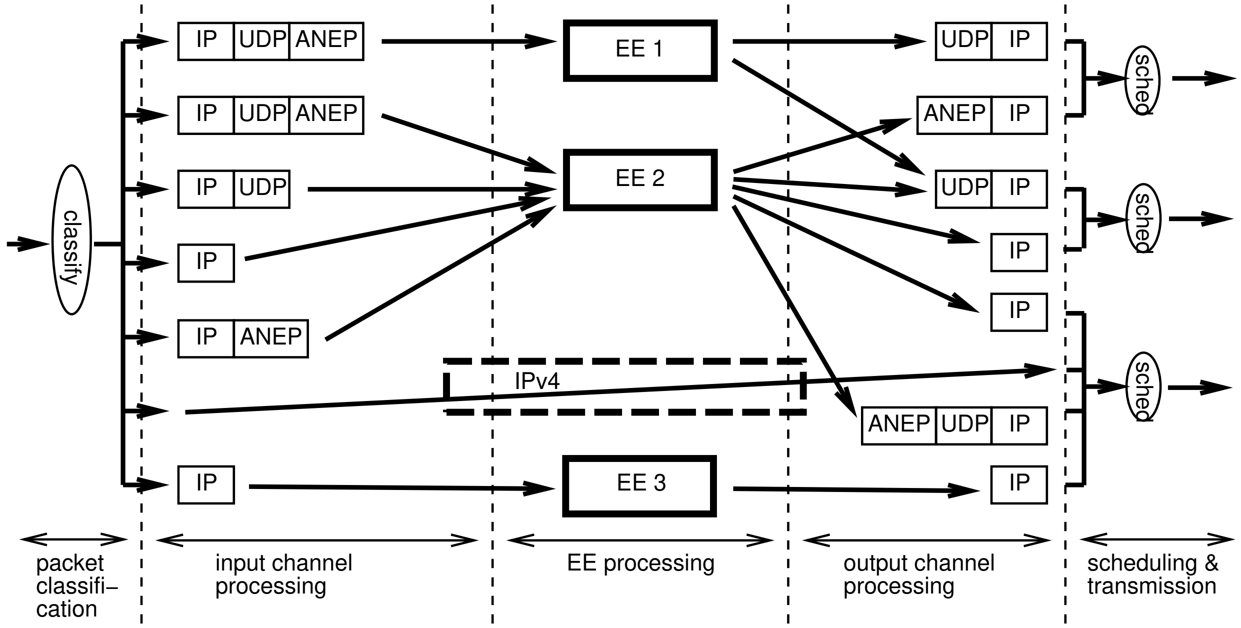

Figure 2: Packet flow through an active node (Link-level protocols omitted to save space)

other traffic. For example, they ensure that a misbehaving execution environment cannot use up all of a node's computation resources. They may also provide for more sophisticated services, such as guaranteeing a specific amount of bandwidth (transmission or processing), or implementing a "fair" service Note that input channels are scheduled only with respect to computation, while output channels have to be scheduled with respect to both computation and transmission. that divides bandwidth more or less equally among a set of classes. As with classification patterns, channel creation requests that specify other than best-effort scheduling are subject to node security policy.

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 EE is present at a node, packets containing a valid ANEP header with a Type ID assigned to that EE (encapsulated in a supported protocol stack) will be routed to channels connected to the indicated EE (provided they do not match some more-specific classification pattern). These default channels are created when the EE starts up. Note that, depending on how they are configured, the ANEP header may occur at various places in the sequence of protocol headers in a packet.

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 above-mentioned "pseudo-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.

4.4 End Systems and Intermediate Systems

Active networks consist of both end systems, which serve as "hosts" for end-user applications, and intermediate systems, which exist primarily to switch packets and to process/interpret/execute them along the way. A feature that distinguishes the active network architecture from the internet architecture is the presence of application-specific functionality (AAs) in the intermediate nodes of the network. Thus both end systems and intermediate systems include NodeOS, execution environments, and active applications. Naturally the two types of systems will have different security and other policies. Note that a single EE with a rudimentary NodeOS may suffice for some end systems. An example would be a handheld mobile unit designed to work with a particular EE (see Section 10 for some discussion of interoperability considerations).

5 Execution Environments and Active Applications

An execution environment defines a virtual machine and "programming interface" that can be accessed or controlled by sending the appropriate coded instructions to the EE in packets. The function of this virtual machine 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. Some EEs implement a "universal" virtual machine, i.e. one that can be programmed to simulate other machines, while others offer a more restricted programming interface, to which the user supplies a simple fixed-size set of parameters.

An Active Application is a program which, when executed by the VM of a particular EE, implements an *end-to-end* service. Thus it is the AA that implements customized services for end-user applications, using the programming interface supplied by the execution environment. Details of how the "code" constituting the AA is loaded into the relevant nodes of the network are determined by the EE; the code may be carried in-band with the packet itself, or installed out-of-band. Out-of-band loading may happen during a separate signaling phase 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 control.

It is expected that most end users will not program the network, any more than most users program their PCs. Rather, applications running in end systems will access active network services by invoking AAs with code provided by the AA developer (which may be the EE developer, or may be a third party).

5.1 Composing AAs

EEs should provide for simultaneous access to more than one service. As a somewhat artificial example, consider an active application that provides mobility, and another that provides a reliable group communication paradigm. Ideally, an end-user should be able to invoke both services simultaneously. However, this is a very challenging problem. An execution environment in which independently-developed mobility and multicast AAs could be composed, without modification, to form a mobile-multicast AA would be a significant achievement.

5.2 Deploying New EEs and AAs

An EE provides a programming interface; to be useful, the semantics of this interface must be sufficiently well specified to allow independent development of active applications that make use of it. Moreover, deployment of a new EE in the active network is a significant undertaking, and implies some trust in the EE by the deploying node administration. 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 active applications control the virtual machine implemented by 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 bytecodes interpreted by the Java Virtual Machine, or the sockets interface).
- The well-known ANEP packet types for which the EE "listens" by default.

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.

Active Application deployment is intended to be a lighter-weight undertaking than EE deployment. Nevertheless, an AA similarly needs to define the "programming interface" interface via which it may be invoked and controlled by end-user applications running in end systems. An AA also needs to specify the set of EEs it "supports", i.e. for which it can supply code (see Section 10).

5.3 Inter-EE Communication

Execution environments are assumed to be 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 the normal channel mechanisms. For example, an active application running in EE X might transmit a packet to EE Y on the same node by sending it to the loopback output channel and including the appropriate ANEP header to ensure that the packet is demultiplexed to Y.

6 NodeOS

The NodeOS is a layer operating between the EEs and the underlying physical resources (transmission bandwidth, processor cycles, and storage). Its existence arises from the need to simultaneously support multiple EEs, the need to provide some basic safety guarantees between EEs, and to guarantee a certain base level of functionality that is common to every active node. This common functionality includes implementing the channels that carry packets to and from underlying communication substrates (including protocol processing), routing packets between channels and EEs *within* the active node, mediating access to node resources such as transmission and computing bandwidth, and optionally supporting common services such as routing.

The functionality exported by the NodeOS to the EEs is specified in a separate document [NodeOS].

6.1 Abstractions

The NodeOS provides EEs with access to node resources. There are three primary resource abstractions: *threads*, *memory*, and *channels*. The thread and memory abstractions are similar to those of traditional operating systems, while channels implement lower-level and/or legacy protocol processing along with packet transmission and reception. The *flow* abstraction serves as the basic unit of accounting; all three resources are allocated to or scheduled on behalf of flows. Thus the NodeOS may associate with a flow a group of concurrently-executing threads, a pool of buffers, and a set of input and output channels.

All requests are made to the NodeOS on behalf of a particular *principal*. Each request must be accompanied by or reference credentials sufficient to verify that it originated from a principle authorized by the node and/or EE security policy to perform receive the requested service or resource. The NodeOS relies on the security policy enforcement engine and the security policy database to authenticate and authorize requests.

6.2 Trust Relationships

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.

7 Interfaces

The major interfaces of the node architecture are: between the EE(s) and the NodeOS; between EEs and active applications; and between active applications and end-system applications. The first is of interest to node vendors and EE developers; the second is of interest to AA developers and EE developers; the third is the concern of AA developers and end application developers.

The NodeOS-EE interface does *not* affect the bits on the wire, and is not directly visible to the end system. 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 EEs. A functional specification of the NodeOS interface is given in a companion document [NodeOS].

The interface between the EE and the active application is by definition EE-specific, and thus reflects the "programming paradigm" of the EE. The interface between AA and end application reflects the nature of the service implemented by the AA using the EE. Some EEs (and AAs) may offer services that are accessible via a narrow, language-independent interface (a la sockets); others may require that the active application code be 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.

Note that two of these three interfaces are hidden from the end-user applications, and thus are (relatively) free to evolve and change over time, without requiring modification of code deployed in end systems.

8 Implications for High-Speed Hardware

The node architecture described here maps more or less straightforwardly to a machine with a conventional workstation architecture. However, the capacity of such machines for switching packets at high speeds is limited by their internal bus architecture. In this section we consider implications of the architecture for high-speed, high-capacity network nodes.

Modern high-speed routers are typically built around a high-capacity switch fabric, which connects input port cards to output port cards (Figure 3a). Packets are parsed and validated on the input port card, the

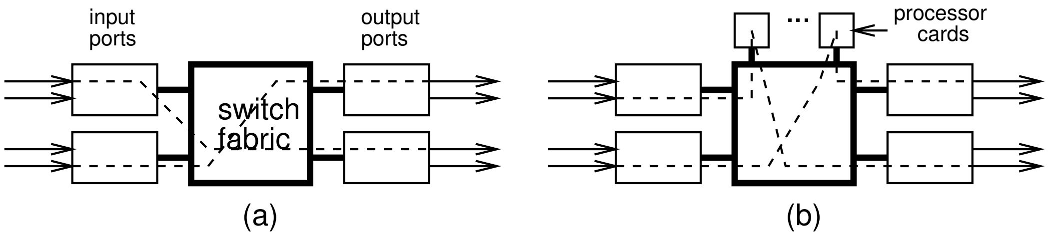

Figure 3: Block Diagram of High-Speed Switch/Router

destination address is looked up in a forwarding table to determine the destination output port card, and the packet is transferred to the output port across the switch. The output port card handles encapsulation for the link and queues (and possibly schedules) the packet for transmission.

One approach to realizing the architectural framework in such a context is to connect dedicated computing modules to the switch and use them to host the execution environments (Figure 3b). Thus packets requiring active processing by an EE are routed to a processor card where the appropriate EE is running, and from there to the appropriate output port. Referring to the packet flow diagram in Figure 2, the first trip through the switch corresponds to crossing the boundary (indicated by a dashed line) between the "input channel processing" and "EE processing" sections of the diagram, and the second occurs when the packet crosses into the "output channel processing" section. Thus input- and output channel processing can in theory occur on the port cards. Cut-through channels allow packets to bypass the processing cards and be handled with one pass through the switch.

This approach scales by adding more processing cards and port cards. However, the NodeOS in this case becomes a *distributed* operating system, with all the additional challenges that that entails. Specifically, any global state store must be shared across multiple processing cards, imposing additional synchronization overhead to access it. Techniques to reduce the performance impact of distribution on EEs and AAs are an important topic for further study.

One disadvantage of this mapping is that EE processing is physically separated from output processing, particularly scheduling. In order to manipulate output queues, for example, either the queues have to be maintained at the computing modules, or the NodeOS must provide methods to examine and modify the queue “remotely”. (Queues associated with cut-through channels can of course be maintained on output cards and will not be manipulated.)

An alternative mapping splits the EE processing between the input and output ports, making it possible to process all packets with a single pass through the switch. After input channel processing, the EE begins processing on the input port card (this approach assumes that adequate computing resources are available on the port cards). If the packet is to be forwarded, the packet *along with its AA context* are sent through the switch to the appropriate output port, where EE processing would continue. To support this mapping, the NodeOS must expose the existence of input and output port contexts and support an “establish output context” system call. The effect of this call is to wrap up the invoking computation and associated state and send it through the switch to the output port, where it resumes execution. This approach requires cooperation from EEs. It has the same challenges of state-distribution as the first approach.

9 Network Architecture Considerations

Essentially all aspects of the global active network service “seen” by an end-user are determined by the combination of the execution environment and the active application, and are thus beyond the scope of this document. This section deals with a few aspects of the global network architecture that do relate to the node architecture, namely information that must be interpreted by both EE/AA/end user and node OS, and mechanisms that can be shared across EEs for global efficiency. First, however, we consider the question of robustness of the active network as it relates to the stability of individual active nodes.

9.1 Robustness and Resource Conservation

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 should not be possible for a malfunctioning or malicious EE or user to “starve” other users and prevent them from receiving adequate resources. To this end, It is desirable to delineate the division of responsibility between the NodeOS and the EEs/AAs with respect to resource management.

As the arbiter of access to individual node resources, the NodeOS must provide allocation and scheduling mechanisms to prevent EEs/AAs/users from depriving other EEs/AAs/users of access to those resources. If resource allocation options go beyond “fair share” allocation to provide specific quantitative guarantees —say, to a privileged flow— waste can occur if reserved resources are not available to others when they are not being used by that principal. Thus, all scheduling algorithms must be 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. However, any mechanisms to do so are tied to the network service and are therefore the responsibility of the EE and AA.

9.2 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.

9.2.1 Quantifying Resource Needs

As discussed above, information about resource demands and usage may need to pass between the end 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.

9.2.2 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.

9.3 Common Objects and Services

The Node OS may, in the interest of efficiency, support certain common abstractions that can be used in any EE. An important example of such an abstraction is a routing table, which maps addresses to next hops (output channels). Rather than requiring each EE 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 EE to create its own instances of routing tables and update and access them. Whether such an instance would be accessible to other EEs should be under the control of the creating execution 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, EEs 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 AA to handle the loading explicitly [SANE].

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

10 Interoperability

This section discusses implications of the architecture for interoperability.

10.1 Requirements for Interoperability

Given a network in which AAs and thus network services can be dynamically developed deployed, it is worth considering the minimum requirements for interoperability.

Consider the simplest case, in which an application program P running in end system X needs to send information to a program Q in end system Y, making use of a particular active application. For simplicity let us assume that there is exactly one path between X and Y, and that it includes nodes A0, A1 and A2. Also, for the purposes of this discussion we assume that everybody trusts everybody else.

In the straightforward case X, Y, A0, A1 and A2 all run the same EE. The appropriate AA is invoked by P and is loaded (if necessary) and executes on all five active nodes. P can be coded to emit packets for a specific EE.

Now consider a case where no common EE exists at all nodes; rather, E0 runs at A0, E1 at A1, and E2 at A2. If the active application can function with "partial activation", P can emit code for, say, E1, and active functionality will be installed only at at node A1. This requires also that P be able to supply adequate EE-independent information for A0 and A2 to forward the packet even though they do not support the referenced EE.

It is conceivable that P might be constructed to include code for all three EEs in each packet. The requires that the AA "support" all three EEs and include the appropriate "code" in each packet. This has the basic problem that ANEP, as currently defined, does not provide any such "pick one of these EEs" semantics. Moreover, in the general case, an active application must "support" a large enough set of EEs so that every node encountered will support one of them. The amount of overhead in each packet is linear in the size of this set. Finally, this approach does nothing to discourage proliferation of EEs. It is therefore discouraged. The preferred approach to interoperability is to encourage the universal deployment of a small number of EEs, and to code active applications to use only one of those EEs at a time. (Note that AAs may still "support" multiple EEs, but the intent is that they would use only one at a time.)

10.2 Active Processing in the Internet

[To be added. Describe how packets may be processed hop-by-hop, including interaction between ANEP and higher-level protocols.]

10.3 Evolution Path

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.

11 Acknowledgements

This document is the result of discussions and contributions from the DARPA active networks research community. A number of people have contributed to this and earlier versions of this draft. Larry Peterson contributed much of the NodeOS section, 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 John Guttag, Gary Minden, Jonathan Turner, 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.

12 Editor's Address

Ken Calvert
Department of Computer Science
773 Anderson Hall
Lexington, KY 40506-0046
Atlanta, GA 30332-0280
Phone: +1.606.257.6745
Fax: +1.606.323.1971
`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.

[PLAN] M. Hicks et al, The Plan System for Building Active Networks, http://www.cis.upenn.edu/ switchware/PLAN/plan-system.ps, February 1998.

[SecArch] S. Murphy, ed. Security Architecture Draft. AN Working Group Draft, 1998.

[NodeOS] L. Peterson, ed. NodeOS Interface Specification. AN Working Group Draft, July, 1999.

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

[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/`.