Composable Services for Active Networks

AN Composable Services Working Group

September 1998

Abstract

A *composition method* provides a means to specify a *composite service* constructed from building blocks called *components*. This document focuses on composition of services to be executed in an active network within a single execution environment. The purpose is to identify features common to all composition methods, present examples of composition methods, and outline larger issues that may be amenable to common solutions.

1 Overview of Document

A *composition method* provides a means to specify a *composite service* constructed from building blocks called *components*. As such, the realization of a composition method is a programming language whose primary purpose is to operate on components, but may have additional language capabilities (e.g., arithmetic, variable declarations). The method and language used to create the components are considered orthogonal to the composition method and are not discussed in detail here.

The emphasis of this document is composition of services to be executed in an active network *within* a single execution environment (EE). Composition within an execution environment is largely an autonomous decision. This document, therefore, does not give a prescription about how composition should be accomplished. Instead the purpose is threefold: (1) to enumerate features that will present, in some form, in any composition method, (2) to provide examples of composition methods discussed in terms of the common features and (3) to outline common issues that go beyond basic functionality and may be amenable to shared solutions. The reader is assumed to be familiar with active networking in general [8] and the architectural framework being developed by the architecture working group [3].

This document includes:

- Terminology to discuss composition (Section 2).
- A list of reference examples of composite service behavior (Section 3). A developer of a composition method might use this as a checklist to determine whether the method satisfies some of the range of desired composition capabilities. Naturally, this list of reference examples is not a complete specification of desired functionality.
- A list of attributes of a good composition method (Section 4). This list emphasizes networking-specific concerns that are likely to influence the success of a composition method.

- A list of features that will be present, in some form, in any composition method (Section 5). This list is intended to support a common method of describing the capabilities of a composition method, to ease comparisons. For example, a user who is selecting an EE for application development might compare the composition support in different EEs using this list of properties.
- A set of examples of composition methods (Section 6). This captures part of the current state-of-the-art in composition methods.
- A discussion of issues that each composition method will need to address, beyond the basic issue of providing mechanisms to create composite services (Section 7). For example, this includes a discussion of the relationship between security and composition. Solutions to problems in these areas may be shared across different composition methods.
- A brief discussion of a straightforward method to achieve cross-EE composition (Section 8).

1.1 Status of Memo

The starting point for this document was the rough consensus of the participants in the composable services working group at the Active Networking Workshop in Tucson, AZ, March 9-10, 1998. A second working group meeting was held at the Active Networking Workshop in Atlanta, GA, July 16-17, 1998. Comments are solicited and should be directed to the appropriate mailing lists (Section 10).

This development of this document was significantly influenced by the book "Programming Languages: Design and Implementation," Pratt and Zelkowitz, Third Edition, 1996.

2 Terminology

The following terms will be used to discuss composition.

- **Component**

  A component is an entity that has a well-defined interface and behavior, as defined by the component developer. Components form the basic building block out of which services are constructed. A component must have both a specification and an implementation. The specification includes the method used to refer to the component, the interface used to access the component, and a description of the behavior of the component. The implementation realizes the interface and behavior given in the specification. A component may have rules that dictate, based on semantics, how it may be used. By design, this definition admits a wide range of options for components, from simple programming language statements to complex behaviors.

- **Service**

  A service is an entity that has a well-defined interface and behavior, as defined by the service developer. The important distinction between a service and a component is that a service is visible (i.e., capable of being referenced) by a user. Some components may be services. Typically, services are created by putting together multiple components using the sequence control mechanisms of a composition method. Such services will also be called *composite services*.

- **Composition method**

  A composition method is the syntax and semantics for creating services from components. This includes the specification, but not the implementation, of components.

3 Reference Examples

The examples given in this section are intended to serve two purposes. First, they define (by example) what is meant by composable services and therefore give the reader something concrete to ground the rest of the (more abstract) discussion in this document. Second, they provide the developer of a composition method with an incomplete list of desired functionality.

3.1 Forwarding with Priority-based Packet Discard

One category of composition involves enhancing a "base" service with specialized functionality. For example, consider a forwarding service whose default behavior is to drop the arriving packet when the queue for the outgoing interface is full. Suppose the service has been designed to support replacing the default behavior with a component that provides an alternative action when the queue is full (e.g., dropping of a lower priority packet from the queue if one exists). A composite service is created by combining the base forwarding service with the alternative action component.

3.2 Packet Filtering

A packet filtering service selects particular packets for additional processing (e.g., logging). A packet filtering service may be created by composing basic filters to specify which packets "match" the selection criteria. This service might support dynamic changes to the set of filters to allow run-time modification of the selection criteria. The components in this case are the basic filters, and the composition method must specify the way the filters are combined to produce a well-defined packet selection criteria.

3.3 Transcoding Gateway

A composite service may be formed from components with substantial capabilities. For example, consider the design of a transcoding gateway service capable of translating information received in format A to format B, and vice versa. The building blocks of this service are an encoder/decoder component for format A, an encoder/decoder component for format B, and a bridge component that provides any intermediate functionality needed between the two coding components.

4 Attributes of a Good Composition Method

The success of a composition method depends on its acceptance for use by service developers. These developers will be influenced by both basic language features and attributes that are particular to networking. Basic language features of importance include clarity and simplicity, support for abstraction, naturalness for the application and ease of debugging and verification [6]. Attributes that are of particular interest for developing network services include:

- Performance

Critical to the success of active networking is the ability to offer services that have good performance. The composition method plays an important role in the performance of composed services. For example, the composition method may dictate which parts of a service can be compiled versus interpreted during execution.

A second component of performance concerns the global impact of using composite services. A composite service should offer good performance to the individual user and place a reasonable burden on the shared resources of the network. This requirement might favor, for example, a composition method that offers a concise packet format for specifying or requesting a composite service [7].

- Ease of security

  Some control over access to composite services will be necessary for security purposes. For example, a particular user may need to be authorized to access a particular composite service. A composition method should support the inclusion of the necessary security mechanisms, for example by preventing side effects that would allow a service to hide behavior. The security implications of composite services are discussed further in Section 7.

- Ease of management

  Once developed and in use, composite services will need to be managed. Management tasks will include tracking the use of services (for accounting and/or billing purposes), identifying the source of a problems that arise, upgrading services, and terminating service execution as needed. A composition method should support the development of network management for composite services either as an integral part of the composition process or as a straightforward post-process.

5 Features of Composition Methods

- Sequence control

  A composition method will have structures for controlling the order of execution of components. These structures are termed *sequence control*. Simple examples of ordering include sequential execution (e.g., execute component A in its entirety followed by component B) and concurrent execution (e.g., execute components A and B concurrently).

  More complex ordering results when components and composite services can interact with one another to create hierarchy in the execution sequence. Various forms of hierarchy include: calling one component from another, calling a component from itself (i.e., recursion) and calling a composite service from another composite service. The last form — encapsulating a composite service to be used as a component in a new composite service — requires a naming mechanism.

- Shared data control

  A composition method will have mechanisms that determine how data is shared between components. Examples of methods for sharing data include: *explicit parameters* that allow the shared data to be passed directly to the component, *shared objects* that allow the definition of data that is accessible by all components involved in sharing, and *inheritance* that allows implicit sharing based on pre-defined relationships between components.

If components can execute in parallel and need to share data, then *message passing* between components provides another data sharing alternative. Note that if shared objects are used for data sharing amongst components executing in parallel, then access control mechanisms will be necessary to ensure the integrity of the data.

- Binding time

  Binding refers to the instantiation (or selection) of an entity from a set of options, as necessary for execution. In the context of composition, the entities that might be selected include the particular components in the composite service. The possibility of different versions of components over time raises interesting binding issues. The options for binding time include at composite service creation time (generally occurring at an end-system) and at composite service execution time (generally occurring at an active node).

- Invocation methods

  The invocation methods are the events that can cause a composite service to be executed. The most common invocation method will be the arrival of a packet at an active node. However, other events may also cause invocation of a composite service. For example, an active node might be configured to execute a neighbor discovery composite service as part of the system startup process.

- Division of functionality

  The specification of a composite service will occur through a combination of content carried in packets and content resident (or dynamically loaded on-demand) at nodes. For example, the packet might carry a program written in a scripting language that includes calls to node-resident service routines [4].

6 Examples of Composition Methods

This section contains examples of composition methods under development within the DARPA active networking program. The emphasis is on the way that each method addresses the features outlined in Section 5. The intention is to illustrate a set of options for composition and facilitate comparison between options.

6.1 Programming Language for Active Networks (PLAN) (Nettles)

PLAN (Programming Language for Active Networks) [4] is a special-purpose domain-specific language for programming Active Packets. Implementations of PLAN using both Java and Caml as the service implementation language can be downloaded from `www.cis.upenn.edu/ ~switchware/PLAN`. Thus far, the most ambitious use of PLAN is to implement the PLANet active internetworking testbed [5]. PLANet is the first active internetwork implementation; in PLANet, all packets are PLAN programs.

PLAN embodies a distributed computational model in which computation proceeds by remote evaluation of PLAN programs on selected nodes throughout the network. For example, data transport in the style of UDP is achieved by sending a PLAN program to the destination with the port and data as arguments. The program is just a function which delivers the data on the given port. In general, PLAN programs replace both the headers and payloads of conventional networks.

On any particular node, the role of PLAN programs is explicitly to invoke and compose services available on the node. The node local semantics of PLAN are loosely based on the first-order typed lambda calculus, and its syntax is derived from the ML language family. Both services and functions defined in PLAN are called using familiar function call syntax. PLAN also supports function definition, statement sequencing, conditionals, and iteration over finite lists. PLAN supports only a limited set of data-types, such as strings, integers, and lists, as well as some special network oriented data-types such as hosts and ports.

Despite its resemblance to a general-purpose programming language, PLAN programs are severely restricted in their expressibility. PLAN provides no mutable data-types (however services may encapsulate mutable data) and it also provides no facilities for defining new data-types. More seriously, recursive functions are not supported, and PLAN programs are guaranteed to terminate as long as the services they invoke terminate. Thus PLAN is not a Turing-complete language.

In the terminology of this document, PLAN's compositional model can be characterized along the following dimensions:

Sequence Control PLAN programs are themselves inherently sequential, although they can invoke services that may not be. Furthermore, the underlying system is free to execute PLAN programs in parallel with other PLAN programs. Since PLAN programs cannot directly manipulate mutable data, such parallel execution is transparent at the PLAN level. PLAN provides conventional language constructs to control sequential execution.

Shared Data Control PLAN does not directly provide shared data. Services, which are implemented in some other language, can (and in practice do) provided shared mutable data, which is managed using the mechanisms provided by the implementation language.

Binding Time PLAN programs are dynamically bound to services when they are executed on a node. PLAN programs are lexically scoped, with services placed in the outer-most scope.

Invocation Methods PLAN packets name a function that is to be invoked when the program is evaluated. The packet also carries the functions arguments. Other PLAN functions and node defined services can be explicitly called from this initial function.

Division of Functionality PLAN makes a strong division of functionality between PLAN-based functionality and services. In particular, a basic goal of the PLAN design was that PLAN programs should not in general require authentication or authorization to execute. Functionality that needs heavy-duty security must be encapsulated by services which are invoked only as needed and which perform the needed authentication or authorization. Similarly, all mutable or shared data is encapsulated by services, and all PLAN data is immutable.

It is important to understand that because of PLAN's deliberately limited nature, implementing non-trivial systems such as PLANet also requires the ability to extend nodes with new services. In our current implementation, these extensions are written in a general purpose language (Caml for PLAN 3.0, Java for PLAN 2.0), and these extensions can be dynamically loaded into nodes. These *Active Extensions* constitute a whole set of composition methods in and of themselves. Since these composition methods are part of the general-purpose programming language used for the extension, we omit discussion of them here.

6.2 Netscript (Florissi and da Silva)

NetScript is a high-level programming language and environment for developing protocols and services in an active network. Programs written in NetScript can be deployed and installed in nodes of an active network to dynamically extend the network with new functions. NetScript views the task of an active node as that of processing packet streams and allocating resources to support this processing. Accordingly, the language supports dynamic construction of packet-stream processing engines, interconnecting these engines and allocating resources to them. This section presents NetScript's approach to composition of active elements. The material presented here is a summary of [2].

NetScript's computational and compositional model is derived from *dataflow*. A dataflow system consists of a collection of components connected together to form a dataflow graph. These components, called boxes in NetScript, process data streams flowing between them. NetScript boxes execute concurrently and their processing is reactive to the specific data arriving through data streams.

The unit of program composition in NetScript is the *box*. A box consists of a zero or more typed input ports (or inports), zero or more output ports (or outports), local state, internal boxes and connections between ports of boxes. The composition mechanism 6 NetScript is 0. One builds composite boxes by connecting together ports of simpler boxes subject to the strong-typing constraints of port types. NetScript boxes communicate with each other through their ports. Typically, message streams flow from box to box and are processed as they traverse the dataflow graph. An optional *event-handler* can be associated with each input port. An event-handler is a sequential program written in Java, C, or other imperative language. Arrival of a packet at a box inport triggers execution of its event-handler, which runs atomically to completion. The event-handler operates on the packet to perform parsing, classification, forwarding, etc.

What makes NetScript suitable for programming protocols in an active network is its support of *dynamic* changes in the dataflow graph; boxes can be added or removed, connected or disconnected at runtime. This property of NetScript enables dynamic composition of active protocols.

In the terminology of this document, NetScript's compositional model can be characterized along the following dimensions:

Sequence control NetScript provides both sequential and parallel composition. A packet that flows through a pipeline of boxes is processed in sequence by each successive box in its path. Any path through the dataflow graph defines implicit order of execution. For example, an incoming TCP packet is processed successively by a datalink box (e.g., Ethernet), an IP box, the TCP box and finally a Socket box. Boxes that do not lie along the same path in the dataflow graph can execute in parallel. For example, a box outport can be connected to the inports of multiple downstream boxes. Each such downstream box proceeds in parallel on arrival of a message from the upstream box.

Shared data control NetScript does not support shared global data structures. A box may encapsulate local private state, but all external access to this state is achieved through formal box interfaces, namely its ports. Lack of global shared state eliminates the need for synchronization primitives (e.g., mutex locks) and simplifies reasoning about NetScript programs.

Binding time NetScript provides both static and dynamic composition (binding) of services. Under static composition, the programmer defines a box by declaring its port signature and statically connecting together internal component boxes. Such a box can be dispatched to

and dynamically installed at NetScript nodes across the network. At installation time, the target NetScript engine creates a runnable instance of this box. This box is dynamically connected to existing boxes at the target engine, as specified in a control script. In addition boxes can also be removed and/or disconnected at runtime, either under remote or local control.

Invocation methods A NetScript program is a reactive system. Arrival of packet streams either from the network or from local user processes triggers invocation of boxes at a NetScript node.

Division of functionality NetScript programs can operate either on passive packet streams or active packet-capsules. Typically, NetScript programs implement protocols and services, active or standardized, that are installed in active nodes and persist until removed. These active services operate on passive packet streams. Packet headers encode the sequence of operations that must be invoked to process them. For example, HTTP packets that flow through a NetScript node are processed successively by Ethernet, IP, TCP and HTTP boxes. Alternatively, a NetScript box can implement an interpreter for a packet language such as PLAN[4]. This box will interpret and execute the program contained in each packet-capsule that is demultiplexed up to it.

6.3 LIANE (Bhattacharjee, Calvert, and Zegura)

Composition in LIANE (Language-Independent Active Network Environment) has two parts. First, the user selects an *underlying program* from amongst those offered by an active node. The underlying program provides a basic service (e.g., forwarding) and includes a set of *processing slots* to be used for customization (e.g., to replace the default forwarding table with a customized forwarding table). Each processing slot is associated with a specific execution point in the underlying program. In the second part of composition, the user selects or provides a set of *injected programs* used to customize the underlying program. Each injected program is "bound" to one or more processing slots. The injected program is eligible for execution when the appropriate slot is reached ("raised"). More than one injected program may be bound to the same slot; the order of execution of instructions belonging to different injected programs bound to the same slot is non-deterministic. This style of composition has advantages with respect to proving properties about the composite service based on properties of the underlying program and preservation of properties by the injection process [1].

In the terminology of this document, the injected programs correspond to components and the underlying program specifies the basic sequence control and part of the execution of the composite. Using the terminology of Section 5, the features of LIANE are the following:

Sequence control The choice of underlying program provides the basic sequence control for a composite. An underlying program can impose linear sequencing by exporting mutually-exclusive slots that are raised in a particular sequence. To enforce strict sequential execution of components, only one injected program can be bound to each slot. Parallel execution can be achieved in two ways: multiple injected programs can be bound to a given slot or multiple slots can be raised at the same time. LIANE does not specify the order or interleaving of bound statements within a slot, thus the statements in different injected programs may be executed concurrently. Further, slots are raised if and only if some slot-specific condition is true. Unless such conditions are mutually exclusive, multiple slots can be raised concurrently.

LIANE allows an injected program to raise events that other injected programs can bind to. Thus, a component can "call" another component. An underlying program with slot bindings (i.e., a composite) can be bound to a slot in another underlying program.

Shared data control Data can be shared between the underlying program and the injected programs, or between multiple injected programs. Both types of data sharing take place through a shared tuple space and through updates to shared variables. Thus, shared data control is achieved through shared objects. There is no notion of inheritance or of explicit parameters passed to components.

There is no explicit message passing or access control imposed by LIANE for shared access during concurrent execution. The injected programs are responsible for maintaining the integrity of their data during concurrent accesses.

Binding time The choice of underlying program is static and done at composite creation time (i.e., when the packet requesting the service is created). An initial choice of injected programs is also done at composite creation time. However, the event binding mechanism can support the dynamic binding of new injected programs at run-time.

Invocation methods Invocation of a composite typically occurs due to the arrival of a packet that requests a particular underlying program and selects or provides a set of injected programs. However, the event paradigm is sufficiently general that a composite could be bound to an event other than a packet arrival, for example a timer interrupt or a variable value exceeding a threshold.

Division of functionality The packet carries the selection of underlying program (chosen from a fixed set resident at the node) and the specification of the injected programs (chosen from those resident at the node or supplied explicitly via code in the packet). Resident at the node are a fixed set of underlying programs and a set of possible injected programs.

6.4 Active Extensions in the ARP Project (Braden)

The basic objective of the Active Reservation Protocol (ARP) project is the "active extension" of signaling protocols. That is, ARP is developing a facility for dynamic composition of new or modified service features with an existing signaling service. With ARP, an initial or "base version" version of the code that implements a particular signaling algorithm can be modified by one or more "functional extensions" (FEs). The base version plus the set of FEs in use for a particular signaling activity define a "version" of the signaling code.

An appropriate version may be selected for each distinguishable signaling activity; for example, for RSVP, an appropriate version may be used for each RSVP session, with the base version as default. New versions may be installed and controlled by network management, which would exert administrative control over the mapping of sessions onto versions by installing of appropriate classifiers. Ultimately, we hope to allow individual customers of the signaling function to be able to develop and dynamically install their own versions; this requires solving the protection and security problems inherent in allowing arbitrary code fragments in the node.

FEs are cumulative in effect, and each successive FE may build upon earlier FEs in the same version. FEs logically form a hierarchy rooted at the base version, since the use of a particular FE generally requires that all FEs "above" it in the hierarchy be in place also.

Implementation of signaling protocols by Java byte code executed in a Java virtual machine allows ARP to build its functional extension (i.e., composition) mechanism upon the standard object-oriented inheritance mechanisms of Java. Thus, the base version and each FE consist of sets of Java classes. The class hierarchy parallels the FE hierarchy: each class in an FE must be a sub-class of a class in a "higher-level" FE or in the base version. Subclass inheritance allows each FE class to extend its immediate super-class by redefining or adding methods and/or fields. Thus, each FE adds additional leaves to the Java class inheritance tree; conversely, the leaves of the inheritance tree are the points at which extension is possible.

Sequence control The extension code uses the sequence control inherent in Java and determined by the base. That is, execution is fundamentally sequential, but with the possibility of multiple threads. Initially, at least, the thread structure will be imposed by the base; extensions will not be allowed to create new threads. This issue will be revisited later.

Shared data control Data sharing uses all of the methods: explicit parameters, shared objects, and inheritance. It is controlled by the existing Java access rules, since all data is stored in the common Java heap. A given FE executing on behalf of an activity will generally share per-activity data with higher-order FEs and the base version; this data will be private to the activity (session). There will also be data to control the allocation of the CPU and other resources to different activities, as well as the maintenance of currently loaded classes; this control data must be protected from the activity execution.

Binding time Normally the base version is expected to be loaded by network management when the node is initialized, while particular versions will be loaded upon demand from arriving signaling protocol packets. However, neither choice is mandatory; the ARP mechanisms should support demand loading of the base version as well as static pre-loading of FEs.

Invocation methods Invocation of the code in a version will be triggered either by arrival of a signaling protocol packet or by expiration of a timer.

Division of functionality A version may be selected by data in a signaling packet (e.g., in a new RSVP protocol object that we have specified), or it may be selected under administrative control, as indicated earlier. Required FE classes will be loaded by an out-of-band mechanism, either from the previous hop or from a central server.

7 Other Issues

Composition of services raises a number of issues that are not specific to the particular composition method. Instead, these issues are likely to arise for any composition method and generic solutions may be applicable across composition methods. It is desirable to identify such orthogonal issues so that solutions can be developed independent of composition methods.

- Node-resident content deployment

  As discussed earlier, the specification of a composite service will occur through a combination of content carried in packets and content resident at nodes. This raises the issue of how the node-resident content is deployed. Possibilities include off-line loading by the node provider as part of node management and on-demand loading. A variety of techniques are possible for on-demand loading, for example code pulling [9].

- Resource discovery

  Creating or using a composite service requires knowledge of available components or composite services. This raises a number of issues in the area of resource discovery. How are components and composite services named, and how are those names known? How are different versions of the same component or composite service managed? What scoping rules apply for the availability of resources? For example, is the availability of a composite service affected by the part of the network topology where invocation will occur?

- Errors and error handling

  Errors associated with composition may occur during creation or execution of a composite service. Error checking and handling during creation of a composite service is likely to be specific to the composition method and thus is not discussed here. We restrict this discussion to errors that can occur during execution, with the understanding that the frequency of these errors may be affected by error checking at composite service creation time.

  A variety of errors may occur during composite service execution. For example, some part of the expected node-resident content may not be present, even after any on-demand loading mechanisms have been employed. Content expected in the packet may not be present. The execution of a component or composite service may exceed resource limits.

  Related to the issue of errors are the mechanisms and policies for handling errors. For example, when an error occurs, who is notified? If an error occurs in a component, how does that affect the rest of the composite service?

- Security

  Composition raises several security issues. Some composite services will require that the user be authorized to execute the composite service. Some components will require that the composite service developer be authorized to use the components in service creation, or that the composite service carry security associations based on the components involved. The authorization to use components may depend on the details of the resulting composite service. (For example, a service developer or user may be authorized to use components A and B, but not in arbitrary compositions.) The discovery of composition resources may also be subject to security-based access constraints. For example, some components and composite services may require authorization to ask about their availability.

- Accounting

  The use of some components and composite services will need to be monitored for accounting purposes.

8 Cross-EE Composition

The focus of this document is composition of services within a single execution environment. In this section, we briefly describe a straightforward mechanism for creating services whose flow of control includes execution in more than one execution environment.

A service executing in one EE may generate an event that causes invocation of a service in another EE, accomplishing cross-EE composition. For example, an ANTS service may generate a PLAN packet and send that packet to the local node. PLAN will process the ANTS-generated packet just as it would process any other PLAN packet.

9 What is Not Covered

This document gives only brief coverage of the following areas, largely due to lack of experience and consensus in the active networking community with respect to these areas.

- Security issues associated with composition of services.
- Alternate forms of cross-EE service composition.

  For example, should services executing in different EE's be allowed to share data structures, such as routing tables? If so, how will access to these data structures be controlled? Are there other forms of cross-EE service interaction that make sense?

10 Addresses

Comments should be sent to the mailing list:

`ActiveNets_Services@ittc.ukans.edu`

also archived at:

`http://www.ittc.ukans.edu/~anserve/`

or to the editor:

Ellen Zegura
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
Phone: 404-894-1403
Email: `ewz@cc.gatech.edu`

References

[1] S. Bhattacharjee, K. Calvert, and E. Zegura. Reasoning about active network protocols. In *IEEE ICNP'98*, Austin, TX, October 1998.

[2] S. da Silva, D. Florissi, and Y. Yemini. Composing active services in NetScript. DARPA Active Networks Workshop, Tucson, AZ, March 1998.

[3] AN Architecture Working Group. Architectural framework for active networks. Ken Calvert, editor, 1998.

[4] M. Hicks, P. Kakkar, T. Moore, C. Gunter, and S. Nettles. PLAN: A Programming Language for Active Networks. International Conference on Functional Programming (ICFP'98), 1998.

[5] PLANet: An Active Internetwork. Submitted to Infocom'99. Can be found at `www.cis.upenn.edu/ ~switchware/papers/planet.ps`.

[6] T. Pratt and M. Zelkowitz. *Programming Languages: Design and Implementation.* Prentice Hall, Third edition, 1996.

[7] B. Schwartz, W. Zhou, Z. Jackson, W. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks. `http://www.net-tech.bbn.com/smtpkts/smtpkts.ps.gz`, Jan 1998.

[8] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1), 1997.

[9] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.