# Active Networking Means Evolution (or Enhanced Extensibility Required)

Michael Hicks
Computer and Information Science
University of Pennsylvania
mwh@dsl.cis.upenn.edu

Scott Nettles[*]
Electrical and Computer Engineering
University of Texas at Austin
nettles@ece.utexas.edu

## Abstract

The primary goal of active networking is to increase the pace of network evolution. The approach to achieving this goal, as well as the goal of enhancing customizability, is to allow network nodes to be extended by dynamically loaded code. Most active network implementations employ *plug-in extensibility*, a technique for loading code characterized by a concrete, pre-defined abstraction of future change. After giving examples of plug-in extensibility, we argue that while it is flexible and convenient, it is not sufficient to facilitate true evolution of the network. To remedy this problem, we propose the use of *dynamic software updating*. Dynamic software updating reduces the *a priori* assumptions of plug-in extensibility, improving flexibility and eliminating the need to pre-plan extensions. However, this additional flexibility results in additional complexity and creates issues involving validity and security. We discuss these issues, and describe the state-of-the-art in systems that support dynamic software updating, thus framing the problem for researchers developing next-generation active networks.

## 1 Introduction

*Active networks* (AN) are networks whose elements are, in some way, programmable. The idea of AN was developed in 1994 and 1995 during discussions in the broad DARPA research community, and since then a significant number of prototypes (*e.g.* [10, 23, 24, 3, 18, 15]) have emerged. Reviewing the early AN discussions, we find one chief motivation driving the initiation of research into AN: *faster network evolution*. For example, the SwitchWare project proposal, from the University of Pennsylvania, states ([21], p. 1):

> The pace of network evolution (not switch evolution, *network* evolution) proceeds far too slowly. To a large degree this is a function of standardization.

How, then, does active networking address this problem? Early work by Tennenhouse and Wetherall motivates that active networking facilitate evolution by *customization* ([22], p. 2):

> The [active network] programming abstraction provides a powerful platform for user-driven customization of the infrastructure, allowing new services to be deployed at a faster pace than can be sustained by vendor-driven standardization processes.

For the most part, existing AN implementations embrace this philosophy and employ customization as the means to evolution. Usually customizability is achieved through *extensibility*: the network elements may be extended with user (or administrative) code to add or enhance functionality. For example, many systems allow new packet processing code to be dynamically loaded, as in ALIEN [4]

[*]This work was supported by DARPA under Contract #N66001-06-C-852. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

and Netscript [24]. In some systems the extensions reside in the packets themselves, as in PLAN [10], and ANTS [23].

While it is clear that all of these implementations add flexibility to the network by way of extensibility, we believe that no existing system truly solves the problem of slow network evolution. Other authors have cited inadequate resource management and security services as the main inhibitor of active network deployment, but we believe the problem is even more fundamental: no existing system is *flexible enough* to anticipate and accommodate the future needs of the network.

In this paper, we look closely at the extensibility strategy employed by many AN systems with an eye towards network evolution. In particular, we find that implementations at their core rely on *plug-in* extensibility, a strategy for loading code that abstracts the shape of future changes with a pre-defined interface. Plug-ins simply and efficiently support a customizable network service, but they are not flexible enough to support true evolution. Drawing from our own experience with PLAN(et) and related past work, we propose a more flexible alternative, termed *dynamic software updating*, that we believe can much more effectively address the evolution problem.

Our presentation is divided into three parts. In Section 2, we define *plug-in extensibility* and provide two concrete examples of its use, the Linux kernel and the PLANet [12] active internetwork. We then show how many mature AN implementations, including ANTS [23], and Netscript [24], among others, also employ plug-in extensibility. In Section 3, we explain how plug-in extensibility is inherently limited with respect to how a system may change; here we present some concrete experience with PLANet. Finally, in Section 4, we propose an alternative to plug-in extensibility, termed *dynamic software updating*. We explain the benefits and potential problems with this approach, and then point to past and future work that will make it a reality. We conclude in Section 5.

## 2 Plug-in Extensibility

Most AN systems provide network-level extensibility by allowing network nodes to dynamically load new code. This is a popular technique in areas outside of AN as well, including operating systems (OS's), like Linux and Windows, and web browsers. When a new service is needed, some code may be loaded that implements the service. Typically, this code is constrained to match a pre-defined *signature* expected by the service's clients. We refer to this approach as *plug-in extensibility*.

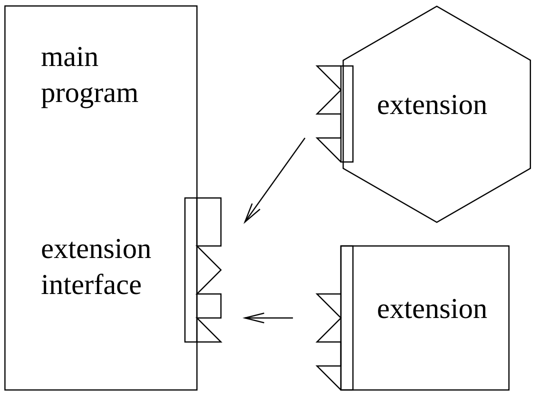

Figure 1: Plug-in extensibility: extensions are "plugged-in" to an extension interface in the running program.

Essentially, plug-in extensibility is a technique that *abstracts* the shape of loadable code. Loaded code is accessed by the running program through an extension interface. Extensions, while internally consisting of arbitrary functionality, may only be accessed by the current program through the extension interface, which does not change with time. This idea is illustrated abstractly in Figure 1. In this section we present some concrete examples of how this works, both to make the ideas concrete, and to demonstrate some problems and limitations of this approach. The impatient reader may skip to Section 3 where we argue that plug-in extensibility is insufficient for evolution.

### 2.1 Linux

The Linux kernel code uses plug-ins extensively; plug-ins are called modules in Linux terminology. Typically the kernel code will perform some test; if the test fails the kernel will load an appropriate module and then retry the test. While the module is being loaded an initialization routine will be called, which alters some of the kernel's *currently visible* data-structures so that the test will succeed the second time. Similarly, when a module is unloaded,

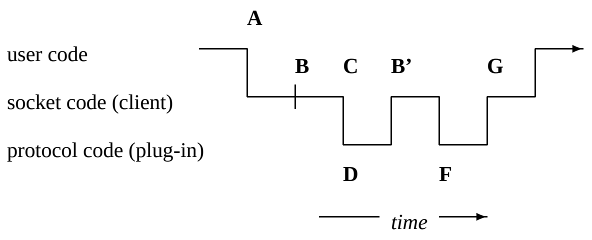

Figure 2: Linux protocol code

a cleanup function is called to remove any vestiges of the module from these data-structures. Web browser plug-ins are implemented using a similar technique.

A specific example for the Linux network stack is shown pictorially in Figure 2, where the code is divided into user code, the socket code, and the plug-in code; time proceeds to the right. The letters label important points in the execution and the text that follows is keyed to these labels.

Suppose a user attempts to open a socket that will use the IPX protocol family (A). The kernel first checks a list of structures, indexed by the protocol family, to see if the IPX handler is present (B). Each handler structure consists of a number of function pointers that implement the functions expected of a network protocol, effectively defining the interface, or *signature*, of an abstract protocol handler. If the IPX handler is not present in this list, then the socket code attempts to load a module that implements it (C). During loading, the IPX module's initialization function will be invoked (D). This function allocates and initializes a new handler structure for its functions, and stores the handler in the kernel's handler list. Afterwards, the socket code checks the list again (B'); when found, the socket code will invoke the handler's socket creation function (F), which will return a new socket structure. The socket code then keeps track of the structure in its file descriptor table, and returns the file descriptor to the user (G). After some period of quiescence, or by user-directive when the handler is not in use, the handler may be unloaded, which will cause the removal of the handler structure from the kernel list.

This technique allows the protocol handling code in the kernel to be extensible. Protocol handlers are treated abstractly by the socket code via the extension interface. In this way, changes may be made on the granularity of protocol handlers—the user and socket portions of the figure will always be fixed, but the plug-in code can change. New protocol handlers for different protocol families may be added, and existing handlers may be removed and replaced with new versions. All loaded handlers must match the the extension interface of the socket code if they are to be accessed correctly.

There are, however, some important limitations. First, it is not possible to change the standard procedure for dealing with protocols. For example, while we could dynamically change the handler signature with some new function types, the old client code will never use them. Similarly, we cannot usefully change the types of current functions, because they will be interpreted using the old types. We could solve this problem if we could alter the client code to take advantage of the new or changed features. But this is not possible because the system has not been programmed to allow the client code to change. Thus, to make these kinds of changes, we would have to recompile and redeploy the system.

Another limitation of Linux protocol modules is that they may not be updated (that is, replaced with an "improved" implementation) while in use. This is because unloading of a module is not allowed while it is use and, more fundamentally, there is no way to transfer the state of the current version of the module to the new version, which would be needed to allow open connections to operate seamlessly. Such a state transfer would have to be anticipated and facilitated by the client code (we demonstrate an example of this idea in the next section). Disallowing updates to active code is probably reasonable for this application, but we will argue that it is not an ideal choice for active networks.

## 2.2 PLANet

Although the details differ between systems, AN implementations make use of plug-in extensibility in much the same way as Linux. As an example of this, we will first focus on our own system, the PLANet [12] active internetwork. Other AN systems will be shown to fall into the same mold in Subsection 2.3.

PLANet is based on a two-level architecture that provides lightweight, but limited, programmability in the packets of the network, and more general-purpose extensibility in the routers. Packet headers are replaced by programs written in a special-purpose language PLAN [10], resulting in much greater flexibility than traditional headers. When packets arrive at a node to be evaluated, their PLAN programs may call node resident *service routines*, which form the second level of the architecture. The service routine namespace is extensible, allowing new service routines to be installed or removed without stopping the execution of the system. This is implemented by dynamically linking code that implements the new service and registering it in a symbol table used by the PLAN interpreter.

The PLANet service-level uses plug-in extensibility. To clarify why this is so, consider the following example. Suppose we want to add a new PLAN service `getRand` that returns a pseudo-randomly generated integer. We must load some new code to implement the service. We present this code piecemeal below, in C.[1] At the core of the new functionality is the function `rand`, which actually generates the random numbers (code not shown):

```
int rand(void);
```

Additionally, we must include an *interface function* `randIfc`, which mediates access between the PLAN interpreter and the actual code. The arguments to all service interface functions include a structure `active_packet_t`, which describes the current PLAN packet, and a list of PLAN values, which are the actual arguments to the service provided by the PLAN program. The `value_t` structure is a tagged union that describes all possible PLAN values. In this case, the arguments are ignored (the list of values should be empty), while the returned `value_t` is tagged with the `INT` tag:

```
value_t *randIfc(active_packet_t *p,
                 list_t *values) {
  value_t *v = malloc(sizeof(value_t));
  v->tag = INT;
  v->val = rand();
  return v;
}
```

Finally, the interface function must be added to the PLAN symbol table, so that it is visible to PLAN programs. This is done via the `register_svc` function, which takes as arguments the name that the function will be referred to by PLAN programs, and a pointer to the interface function. When the new code is loaded, its `init` function will be executed (just as in the Linux protocol handler example), which calls `register_svc` with the name *getRand* and a pointer to the interface function:

```
extern void register_svc(
  char    *planSvcName,
  value_t *(*ifcFun)(active_packet_t *,
                     list_t *));

void init(void) {
  register_svc("getRand",randIfc);
}
```

Why is this plug-in extensibility? The giveaway is the type of `register_svc`. All service routines that may be added and later accessed by PLAN functions must correspond to the type of `ifcFun`, which is essentially the extensibility interface for the system. In the Linux protocol code, plug-ins are allowed to be new or improved protocol handlers; for PLANet, plug-ins are PLAN services. Services are not the *only* kind of plug-in in PLANet, as we shall see in the coming sections.

## 2.3 Other Active Networks

Just because PLANet uses plug-in extensibility does not imply that all active network approaches are so limited. For the remainder of this section we touch on some of the more mature AN implementations and show that while the makeup of plug-ins differ, all systems use plug-in extensibility. Typically, systems fall into two categories, those based on *active packets* (or capsules), and those based on *active extensions*. For the former, we concentrate on the Active Network Transport System [23] (ANTS) as the representative system, and for the latter we focus on Netscript [24].

### 2.3.1 ANTS and Active Packet Systems

ANTS is similar to PLANet in that it makes use of packets that (logically) contain programs, termed *capsules*. These capsules are written in Java, and dynamically linked into the ANTS implementation with access to a node programming interface including a subset of the JVM libraries and some additional utilities. Rather than carry the packet code in-band, the code occurs in the packet *by reference*. In the common case, the reference will be to code present

[1]The code examples shown here are in C, but in reality, PLANet uses the type-safe language Ocaml [14] for its implementation.

```
typedef struct {
  void f(fields_t *pkt_fields,
         list_t *args,
         void *payload);
} capsule_t;
```

Figure 3: Packet program abstract signature

in the node's code cache; otherwise, a built-in distribution system will retrieve and dynamically link the code.

In ANTS, the plug-ins are the capsule programs themselves. Each capsule plug-in essentially has the signature shown in Figure 3: all packet programs are a single function whose arguments are the packet's fixed fields (like source and destination address), some protocol-specific arguments (like a sequence number, or flow identifier), and finally a generic payload.

Because capsules are the only plug-in, much of the ANTS system is not subject to change; this includes the code distribution service, the entire node programming interface, the packet marshalling and unmarshalling code (for the protocol-specific arguments), the code cache, the security enforcement mechanisms, etc. The only way that an ANTS node can change with time is by loading different capsule programs. If some aspect of the node programming interface, or the distribution protocol, needs to be changed, then the nodes would have to be changed, recompiled, and redeployed.

In general, this reasoning applies to other active packet systems, like SmartPackets [19], PAN [17], and the packet programs of PLANet.

### 2.3.2 Netscript and Active Extension Systems

Netscript is a system for writing composable protocol processors. The central abstraction in Netscript is called a *box*, which is conceptually some piece of code with a number of *in-ports* and *out-ports*; in-ports receive incoming packets, and out-ports transmit packets. Boxes are dynamically loaded and connected together by these ports to form modular protocol processing units.

In Netscript, the form of plug-in is the box: all loadable objects must subclass the `Box` class (Netscript uses Java as its underlying implementation). Because the majority of a system built with Netscript is composed of boxes, much of the system is subject to change. Exceptions include the Java libraries (the box programming interface, in some sense), the box-loading system, and the top-level packet filter. However, some boxes are in essence unchangeable because they encapsulate state, and thus cannot be safely replaced, along the same lines as the Linux example.

An interesting way to improve the evolutionary ability of a Netscript system would be to wrap the library routines in boxes. For example, we could create a `HashBox` box to represent the `Hashtable` class. To create a new "hashtable", we would send a message to one of the box's in-ports, and it would emit an object on one of its out-ports. By sending the object and some arguments through various `Hashbox` in-ports, and extracting results from selected out-ports, we simulate the normal function call semantics. The benefit would be that an improved hashtable implementation could replace the old one, if needed. However, this technique is clearly tedious and error-prone, and thus not really practical. We will return to this idea in the next section.

A number of other systems are similar in spirit to Netscript. ALIEN [3] was primarily designed for building modular networking code. CANES [15] makes use of program templates (called the *underlying programs*) parameterized by some *slots* that contain user-defined code. In all cases, the elements that may be loaded are limited in the same way as Netscript boxes.

### 2.3.3 Other systems

Some AN systems do not rely on dynamic linking as their underlying means of extensibility. Instead, they use a more traditional hardware-based, process model for extensions. For example, ABLE [18] is an architecture whose extensions are processes spawned by the node's session manager.

In these systems, plug-ins are essentially whole programs whose extensibility interface consists of the allowed IPC mechanisms to the rest of the system. Just as plug-ins are limited by the programming interface with which they may be called, these programs are limited by their IPC interface.

```
pktQ_t packetQ = global packet queue
void queue_packet (active_packet_t *p) {
  queue p onto packetQ;
}
active_packet_t *dequeue_packet () {
  if queue length is > 0
    dequeue top element from packetQ;
  else
    throw an exception
}
...
```

Figure 4: A Simple Queue

## 3 Why Plug-in's are Insufficient

Plug-in extensibility is convenient, useful, and widespread. Despite this, we believe that if AN is to facilitate substantial network evolution we must go beyond plug-ins. In this section we argue why this is so, and in the next section we propose how to do better.

### 3.1 Limitations of Plug-ins

Plug-ins are convenient because they abstract the kinds of changes that may be made in the future, and thus give the current code an interface to deal with those changes. In the Linux case, the socket code does not care *what* code it is calling, only that it will perform the proper *kind* of function (like setting up a socket object and returning it). Similarly with PLAN services, the caller (the PLAN interpreter) only cares that the service function performs some action with the given arguments and returns a PLAN value.

However, to create a system that is robust to long-term change, as is the goal in active networking, we need to minimize our assumptions about the system. Concretely, we want to minimize the size of the *unchangeable program*. This is the part of the program that is not made of plug-ins, and therefore is not amenable to change. The larger this part of the program, the more likely that some future demand will be impossible to accommodate. To make this point more concrete, we consider a practical example that we encountered with PLANet.

```
/* Type of the queue plug-in */
typedef struct {
  void (*q)(active_packet_t *);
  active_packet_t *(*dq)(void);
  ...
} queue_plugin_t;
```

Figure 5: Queue Plug-in Type

### 3.2 Evolving PLANet's Packet Queue

During the development of PLANet, we decided that a useful AN application would be to allow administrators to change their queuing discipline on demand, to meet current network conditions.[2] In particular, we wanted to be able to change from a single FIFO queue shared by all devices to a set of queues, one per device, serviced round-robin to obtain fair-queuing. But we could not do this unless queues could be plugged-in; otherwise, we could not force the current code to use the new implementation. Therefore, at the time, we coded the queuing discipline to be a plug-in in PLANet.

Our initial queuing implementation, which did not anticipate change, is shown in Figure 4, defining queuing operations like `queue_packet`, `dequeue_packet`, etc., to operate on a globally defined queue. This is simple and easy to read.

To make queues a plug-in, we first defined the type of the plug-in, shown in Figure 5. We then created a default implementation, and provided a means to access and change the implementation at runtime, shown in Figure 6. Here, the default queue implementation is created with `defaultQ`. Users of the packet queue access its functionality through the interface functions `queue_packet`, `dequeue_packet`, etc. These functions call the plug-in's functions and return their results. Future replacements are installed using `install_qp`. This setup is almost exactly the same form as the Linux protocol code, but with one difference: `install_qp` transfers the old state (the packets in the queue) to the new implementation, and thus the timing of the change is not limited to when the queue is inactive. All queue replacements are constrained to match the type of

[2]In fact, this application arose out of the need to demonstrate network evolution.

```
/* Global queue */
static queue_plugin_t *q = NULL;

/* Default implementation */
void defaultQ(void) {
  q = malloc(sizeof(queue_plugin_t));
  q->q = ...;
  q->dq = ...;
  ...
}

/* User interface */
void queue_packet (active_packet_t *p) {
  q->q(p);
}
active_packet_t *dequeue_packet () {
  return q->dq();
}
...
/* To load a new queue implementation */
void install_qp (queue_plugin_t *nq) {
  Move packets in old q to new one, then
  q = nq;
}
```

Figure 6: A Complicated Queue

`queue_plugin_t`.

While queues are now dynamically updateable, there are two basic problems. First, we needed to anticipate not just the need for the implementation to evolve, but even the form the evolution should take (that is, the interface of the queue plug-in). Second, the code that is "plug-in enabled" is substantially more complicated and less clear that the code written in the natural way. We can easily imagine that constructing a large system in this way, with many kinds of plug-ins, will result in obfuscated, error-prone code. Or equally likely, we can imagine that programmers will decide to sacrifice the ability to one day extend the code, to make their immediate task easier.

There is a more important problem. In this case, we anticipated that queues should be plugged-in, and coded the system as such. However, *evolution* implies change in ways we cannot anticipate, and thus may not fit our pre-defined mold. For example, the designers of the Internet did not anticipate its current demand for Quality of Service—it was specifically excluded from the design of the best-effort network service (the telephone network already did QoS well). Yet, high demand for QoS is precipitating proposals to change the Internet, including diffserv [1], intserv [2], RSVP [6], etc. Therefore, we feel it is not really reasonable to think we can choose just the right abstractions now and have those choices hold up over many years.

Ideally, we would make *every program component a plug-in*, but without the problems of code obfuscation and fixed interfaces that we saw above. What we really need is a solution that allows more general changes to be made without having to choose the *form* of those changes ahead of time; we shall explore this idea in the next section.

## 4 Dynamic Software Updating

Ideally, we would like to code our system in the manner of the simple queue implementation, but still be able to support evolution by updating components dynamically, as with the plug-in version. Furthermore, evolution would be facilitated if we had more flexibility in how we perform the updating than is typically afforded by plug-ins. For example, we would like to be able to change the type of components at runtime, rather than limiting the replacement type to match the original compile-time type. In this section we argue that rather than just plug-in extensibility, active networks require what we term as *dynamic software updating* to achieve true evolution.

We begin by defining the requirements of dynamic software updating, and their ramifications in terms of the *validity* and *security* of the system. We finish by pointing to some promising past and present efforts to realize dynamic software updating, with the hope that AN researchers will integrate these techniques into their next generation systems.

### 4.1 Dynamic Software Updating

In order to facilitate system *evolution*, we have the following requirements, which comprise *dynamic software updating*:

- *Any* functional part of the system should be alterable at runtime, without requiring anticipation by the programmer. In the queue example, we would be able to code the queue in the simple, straightforward manner, but still change it at runtime.

- Alterations should not be limited to a predefined structure, i.e. component signatures should be changeable, allowing the implementation to evolve as demands change.

  For example, suppose we want the queue to track how many packets have been processed. With queue plug-ins, while we could dynamically add a new queue implementation that counts packets, we could not make this information available; the type of `queue_plugin_t` (Figure 5) constrains all queue replacements to implement exactly the functions listed. Instead, we would like be able to change this type, either to add new functionality, such as to count packets, or to alter the types of existing functions.

- The timing of updates should not be restricted by the system. In the IPX example, we could not unload the module while it was in use to replace it with a better version. In PLANet and other ANs, some components of the system may *always* be in use; for example, the packet queue may always contain some packets. In the queue plug-in code, we dealt with this situation by transferring all packets from the old queue to the new at installation time. We must allow for a similar mechanism when changes are unanticipated, which implies that when new code replaces the old, it must be able to take care of transferring any necessary state.

While a system that meets these requirements will be significantly better equipped to deal with network evolution, the additional flexibility leads to complications. In particular, it becomes more difficult to ensure that a dynamic extension is *valid*, and to *secure* that extension.

### 4.1.1 Validity

Returning to the example of PLANet services, say we wish to update the `register_svc` function to additionally include the PLAN type of the service:

```
extern void register_svc(
  char    *planSvcName,
  value_t *(*ifcFun)(active_packet_t *,
                     list_t *),
  list_t  *argTypes,
  plan_type_t returnType);
```

We have added two additional arguments: `argTypes`, which describes the expected types of the PLAN service function's arguments, and `returnType`, which describes the type of the return value.

This information will be stored in the service symbol table (along with the old information) to type-check the services called by a particular PLAN program. To do so, we have to alter the table's format to include the new fields, which has two implications. First, before the new code that implements `register_svc` can be used, the data in the current symbol table will have to be converted to match the new type.[3] The second implication is that we will have to change the other functions that directly access the table to be compatible with the new type. We cannot sidestep these issues, as we did in the IPX handler case, by waiting for the symbol table to become empty before making the change because it may never become empty.

Now we have several new concerns:

- What about old client code that calls symbol table access functions like `register_svc`? This code will still expect these functions to be of the same type as before. A quick answer would be that all the old code must be updated to use the new functions. However, this is not feasible since other parties may have loaded code that calls these functions, and we may not have access to that code. We therefore need some way to allow the old code to access the new functions.

- When is a reasonable time to make the change? If the node is accessing the table, perhaps in another thread, when the transformation takes place, then changes to the table could be lost or made inconsistent. Thus, we need to time the transformation appropriately, perhaps with assistance from the application.

  To clarify this point, consider that the old version of `register_svc` is running in thread $t_1$ and is just

[3]In general, for large amounts of state, we may be concerned about the time taken to perform the conversion; we may prefer it to happen incrementally rather than all at once. This is especially important for communications systems, like active network nodes, that may have soft-real-time requirements.

about to add a new entry to the table, when in another thread $t_2$ the new version is loaded to replace it. We might naively think that, at the time of update, $t_2$ could translate the state from the old representation to the new and store this in the new module, similar to what we did in the queue example. However, this translation may not correctly include changes made to the state by thread $t_1$. At best, all that will happen is that the change will not be reflected in the new program. At worst, e.g. if the translation begins just after $t_1$ starts to alter the table, the translated table will be inconsistent. Therefore, properly timing the changes to ensure that state is consistent is extremely important.

These questions follow from the general question of what constitutes a *valid* change to a program. That is, when will an update leave a program in a "reasonable" state? Not surprisingly, Gupta has shown that determining the validity of a change is in general undecidable [8]. Therefore, we must rely on structuring our program so as to simplify the question. When we use plug-in extensibility we essentially limit the forms that changes may take, and can therefore more easily understand their effect on the system. We must similarly learn how to formulate sound methodologies, preferably with a formal basis, for ensuring validity when making the more sophisticated kinds of changes mentioned here. Because the methodology used depends on the kind of change, we do not want to impose a general set of restrictions. However, having some notion, whether enforced by the system or not, of what constitutes a valid change is critical to the practical use of the system.

#### 4.1.2 Security

A topic related to validity is security. Assuming we can avoid integrity failures by using type-safe dynamic linking (in a language like Java, or Typed Assembly Language [7, 16]), we must still worry because the greater a system's flexibility, the greater the risk of problems. For example, in the current plug-in version of PLANet, there is generally no possibility of new code maliciously or inadvertently preventing the operation of the packet processing loop since this code was not coded to expect possible change. However, when we add the ability to change any part of the system, as proposed above, this property is no longer guaranteed, constituting a significant threat to node security. A related problem is information security. That is, certain services may contain private information that should not be made available to other services. However, if complete access to those services is available to *all* new or updated code, then there can be no privacy.

Both problems may be avoided via *module thinning* [3, 11], a technique whereby new code may access old code commensurate with its level of privilege. For example, a routing table service in the node may allow anyone to read the table, but only certain individuals to write to it. This can be controlled by thinning the table-writing function from the environment of inappropriately-privileged code.

In general, while the total space of threats to security increases with flexibility, the need to deal with these threats is application-dependent. For example, the security of a personal computer operating system is probably less important than that of a generally-available active network node.

## 4.2 Achieving Dynamic Software Updating

Given the evolutionary benefits of dynamic software updating over plug-in extensibility, how can we implement it and mitigate its additional problems of validity and security? In this subsection, we present some of the more recent work in the area and point to some promising approaches. In general, no existing system meets all of the requirements we have mentioned. We hope to draw from this work to arrive at a more comprehensive solution [9].

#### 4.2.1 Erlang

Erlang [5] is a dynamically-typed, concurrent, purely functional programming language designed for building long-running telecommunications systems. It comes with language-level and library support for the dynamic update of program modules. If the old module is active when the update occurs, then it continues to be used until called from an external source. If any call to a procedure is fully-qualified (i.e. function `iter` in module `M` syntactically specifies its recursive call as `M.iter()` rather than simply `iter()`) then the new version of the function is called, if it is available. Only two versions of code may be available in the system at any given time; the current old version of code must be explicitly deleted (if any exists) before new code may be loaded, and certain library routines may be used to detect if the old code is still in use.

In Erlang, we could code our system in a straightforward manner but still replace its components at runtime. However, Erlang does not provide any automated support for ensuring validity or security—the programmer must ensure reasonable timing and shape of updates. On the other hand, Erlang has language features that make this process more straightforward: 1) all data is write-once (no mutation), and 2) all thread-communication occurs via message passing. This effectively means that only one thread will ever "change" long-lived data (by passing a modified copy to its recursive call), and all other threads may only access this data in some distilled form via message passing. In this way, essentially *all* function calls to other modules are stateless: the state carried around by a thread is in its argument list, and the only way to get at state managed by another thread is to pass it a message and receive its response (which is different than a function call).

In general, we believe that the Erlang model of dynamic software updating is a good step towards facilitating evolution: it is simple and yet very flexible. In future work [9], we plan to generalize the updating notions in Erlang to less restricted environments (i.e., ones that allow mutation), to add further automated support (i.e. load-time type-checking), and to better formalize the programming patterns necessary to preserve correctness. We have begun to implement this sort of model in Typed Assembly Language [7].

### 4.2.2 Dynamic C++ classes

Hjalmtysson and Gray have designed and implemented mechanisms for the dynamic update of classes in C++ [13]. Their implementation requires the programmer to specially code classes that may be dynamically replaced using a proxy class `Dynamic`. `Dynamic` allows objects of multiple versions of a dynamic class to coexist: it maintains a pointer to the most recent version of a class, directing constructor calls to that class, while instance methods are executed by the class that actually created the object.

This project demonstrates the appeal of an object-oriented approach to dynamic software updating: by using instance methods, an instance's operations are consistent throughout its lifetime, even if a newer version of its class is loaded later. However, determining which set of static methods to use at runtime may be difficult, so the system prevents their replacement. This may be overly restrictive, as all conceptually global data must be anticipated at deployment.

The chief drawback of this approach for our purposes is the lack of safety of the C++ language. While the authors state that the loading of new classes preserves type safety if it exists, C++'s lack of strong typing makes it inappropriate for loading untrusted code.

### 4.2.3 PODUS

PODUS [20] (Procedured-Oriented Dynamic Update System), developed by Mark Segal and Ophir Frieder, provides for the incremental update of procedures in a running program. Multiple versions of a procedure may coexist, and updates are automatically delayed until they are syntactically and semantically sound (as determined by the compiler and programmer, respectively). This is in contrast to Erlang and Dynamic C++ classes, which allow updates to occur at any time.

Updates are only permitted for non-*active* procedures. Syntactically active procedures are those that are on the runtime stack, and/or may be called by the new version of a procedure to be updated. Semantically related procedures are defined by the programmer as having some non-syntactic interdependency. Thus, if a procedure $A$ is currently active, and is semantically related to procedure $B$, then $B$ is considered semantically active.

Updates to procedures are allowed to change type, as long as special *interprocedures* are provided to mediate access; interprocedures are stubs that have the old type, perform some translation, and then call the new function at its new type. This is especially useful in AN, since code originates from different sources. Library functions may be updated to new interfaces even though their client code may not be available for change.

## 5 Conclusions

Active network research to date has made great strides in defining a *customizable* network architecture; most projects add non-trivial flexibility to the use and administration of the network. However, no existing system truly solves the problem of slow, long-term network evolution, because the form of future updates is too restricted. In particular, most systems use plug-in extensibility as their means of loading code. In this paper, we have identified some of the shortcomings of plug-in extensibility with regard to system evolution, and have proposed a way to ease those restrictions in the form of dynamic software updating.

While the topic is quite old, research into dynamic software updating is really in its early stages, and more experience is needed. Because of its applicability to many areas outside of active networks, we hope that more advances will be made in the coming years, to allow system engineers to construct systems simply that are nonetheless updateable. Work is especially needed to ensure that updates are applied to these systems in a safe and secure manner. We feel that this is one of the most important problems facing the active networking community today and plan to vigorously pursue it in future work [9].

## References

[1] Ietf differentiated services working group, 1999. `http://www.ietf.org/html.charters/diffserv-charter.html`.

[2] Ietf integrated services working group, 1999. `http://www.ietf.org/html.charters/intserv-charter.html`.

[3] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks*. PhD thesis, University of Pennsylvania, September 1998.

[4] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.

[5] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[6] R. Braden, L. Zhang, S. Berson, S. Herzog, and S Jamin. Resource reservation protocol (RSVP) - version 1 functional specification, March 1996.

[7] Karl Crary, Michael Hicks, and Stephanie Weirich. Safe and flexible dynamic linking of native code. Submitted for publication, available at `http://www.cis.upenn.edu/˜mwh/taldynlink.ps.gz`, March 2000.

[8] Deepak Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[9] Michael Hicks. Dynamic software updating. Technical report, Department of Computer and Information Science, University of Pennsylvania, October 1999. Thesis proposal. Available at `http://www.cis.upenn.edu/˜mwh/proposal.ps`.

[10] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, September 1998.

[11] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999.

[12] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, March 1999.

[13] Gisli Hjalmtysson and Robert Gray. Dynamic C++ Classes, a lightweight mechanism to update code in

a running program. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[14] Xavier Leroy. *The Objective Caml System, Release 2.02*, 1999. Available at `http://pauillac.inria.fr/ocaml`.

[15] S. Merugu, S. Bhattacharjee, Y. Chae, M. Sanders, K. Calvert, and E. Zegura. Bowman and CANEs: Implementation of an active network. In *Proceedings of the Thirty-seventh annual Allerton Conference on Communication, Control and Computing*, September 1999.

[16] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. In *Proceedings of the Principles of Programming Languages*, pages 85–97, January 1998.

[17] E. Nygren, S. Garland, and M. F. Kaashoek. PAN: A high-performance active network node supporting multiple mobile code systems, March 1999.

[18] Danny Raz and Yuvall Shavitt. An active network approach for efficient network management. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 220–231. Springer-Verlag, June 1999.

[19] B. Schwartz, A. Jackson, T. Strayer, W. Zhou, R. Rockwell, and C. Partridge. Smart packets for active networks. In *Proceedings of the 1999 IEEE 2nd Conference on Open Architectures and Network Programming (OPENARCH'99)*, March 1999.

[20] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, pages 53–65, March 1993.

[21] J. M. Smith, D. J. Farber, C. A. Gunter, S. M. Nettles, Mark E. Segal, W. D. Sincoskie, D. C. Feldmeier, and D. Scott Alexander. SwitchWare: Towards a 21st century network infrastructure. Technical report, University of Pennsylvania, 1997.

[22] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996.

[23] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.

[24] Yechim Yemini and Sushil da Silva. Towards Programmable Networks. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, September 1996.