The SwitchWare Active Network Architecture

D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith*
University of Pennsylvania

July 7, 1998

Abstract

Active networks must balance the flexibility of a programmable network infrastructure against the safety and security requirements inherent in sharing that infrastructure. Furthermore, this balance must be achieved while maintaining the usability of the network. The SwitchWare active network architecture is a novel approach to achieving this balance using three layers: *active packets*, which contain mobile programs that replace traditional packets; *active extensions*, which provide services on the network elements, and which can be dynamically loaded, and; a secure *active router infrastructure*, which forms a high integrity base upon which the security of the other layers depends. In addition to integrity-checking and cryptography-based authentication, security in our architecture depends heavily on verification techniques from programming languages, such as strong type checking.

1 Introduction

The IP Internet provides a 'virtual infrastructure' that creates the illusion of network-wide addresses and packet formats while in fact providing these interfaces using a variety of real networks. IP interoperability works because IP was designed to require minimal subnetwork capabilities, hence its ability to run (albeit slowly!) on 'two tin cans and a string'. Given the success of IP, the lesson that virtual infrastructures can work seems very clear. It is also clear that there are a wide variety of possible virtual infrastructures, and some of these may offer interesting technical advantages. In particular, the choice of the interoperability layer (for IP, the packet format and its addressing scheme) could result in an interoperability layer with a higher level of abstraction, such as a programmable interoperability layer. Active networks are an approach to providing a programmable network infrastructure based on such a programmable interoperability layer. Proposals exist [17] for accessing programmability on a per-user or per-packet basis. These ideas offer considerable power to programmers wishing to create advanced services or test new approaches to providing existing services.

Consider, for example, the problem of negative acknowledgments (NACKs) in a multicast protocol. Figure 1 illustrates a tree of nodes with the source **A** at the root. If nodes **F** and **G** indicate that they did not receive a message, the simple behavior might be for each to send a NACK to **A**, resulting in one or more resends of the missing packet. This poses a scalability risk, as the source could be overwhelmed by NACK's, leading to the condition known as *NACK implosion*.

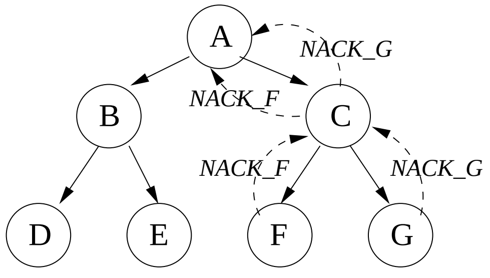

Figure 1: NACK Implosion in a Multicast Tree

With a programmable network infrastructure, an active NACK packet could operate by first checking if a NACK for the same message had preceded it at a node; if so, it would add its sender to a list for retransmission. If it is the first NACK at a node, it would leave a marker indicating which message it was NACKing and proceed towards the root. In this way, an 'as-needed' retransmission of the message would occur.

Of course, such a protocol could be deployed in the current network, but a key question is: *how quickly?* Because the current IP internet is not particularly flexible, it would require changing software on every node in the network—a daunting task. By contast, however, an *active* network is designed with such upgrades in mind. Thus, adding a new protocol is as simple as writing a new router extension or a new active packet program. This speedup in network evolution is one of the primary motivations behind adding programmability to the network infrastructure.

Independent of whether their programmability is offered on a per-packet, per-user, or other basis, active network elements (such as active bridges, switches and routers) must provide facilities for loading and executing programs. Shared network elements will require protection and security, and even single-use network elements derive safety benefits from a well-thought-out model for resource sharing and protection. We believe that the most important knowledge derived from exploratory active network efforts will be an understanding of the tradeoffs between flexibility, security, performance, and usability. The SwitchWare architecture provides a vehicle with which we can evaluate these tradeoffs.

*This paper will appear in IEEE Network Special Issue on Active and Controllable Networks.

2 The SwitchWare Architecture

SwitchWare uses a layered architecture to provide a range of different flexibility, safety and security, performance, and usability tradeoffs. These layers allow us to employ a variety of different approaches to meeting the challenge of providing security in a programmable network, while still gaining the flexibility of programmability and leaving the network usable. These approaches include providing some functionality that is inherently safe and secure, in large part because it is not powerful enough to cause harm, using cryptography-based security to establish and maintain trust relationships, and using verification technologies such as type-checking and program verification to prove that other functionality is safe and secure. As shown in Figure 2, we employ three main layers: active packets, active extensions, and a secure active router infrastructure. We discuss each of these in turn in this section, and more fully elaborate each of them in turn in Sections 4-6.

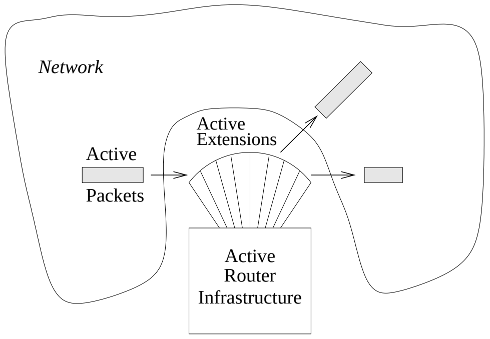

Figure 2: Packet Language and Services

Active packets replace the traditional network packet with a mobile program, consisting of both code and data. The code part of an active packet provides the control function of a traditional packet header, but does so much more flexibly, since it can interact with the environment of the router in a more complex and customizable way than the simple table lookup provided by headers. Similarly, the data in the active packet program replaces the payload of a traditional packet, but provides a customizable structure that can be used by the program. Because even the most performance critical aspect of the network, basic data transport, is done by executing programs, it is crucial that executing these programs be as lightweight as possible.

As discussed in detail in Section 4, we have designed PLAN (Programming Language for Active Networks) to serve as a programming language for active packets. PLAN is very simple and provides a minimum of functionality. Its execution model includes a mechanism for remotely evaluating PLAN programs on other routers; these mechanisms are the means by which PLAN programs transport themselves through the network. PLAN programs are strongly typed to provide safety, and can be statically type-checked before being injected into the network to eliminate the possibility of type errors occurring on remote routers, thus improving usability. In some instances, authentication and other costly checks are unnecessary. PLAN has been designed to avoid the need for such checks in most cases by restricting their actions—for example, a PLAN program cannot by itself manipulate node-resident state. To compensate for these limitations, PLAN programs can call node-resident service routines, which can authenticate or use other more heavyweight mechanisms to provide security on an as-needed basis. Furthermore, PLAN provides a number of different mechanisms for limiting the resources used by an active packet.

We have also experimented with a variant of the ML programming language called Caml [7] as our active packet language. Because Caml does not have the resource limitations described in PLAN, there is a need to provide authentication checks and to provide measures to ensure that security attacks will not succeed. However, in those cases where these checks are necessary anyway, using a single language for both active packets and active extensions provides greater integration. Our initial implementation of SANE uses this facility.

Node-resident extensions form the middle layer of our architecture. They can be dynamically-loaded *active extensions*, or they can be part of the base functionality of the router. They are not mobile: to communicate with other routers they use active packets. Because they are only invoked when needed, there is no inherent need for extensions to be lightweight. They can be written in general purpose-programming languages, although type-safety still plays an important role in this layer. They can use a variety of security mechanisms, including cryptography-based authentication and program verification. Their greater flexibility provides an important trade-off with active packets since complex protocols and systems are implemented in SwitchWare as a mixture of PLAN and router extensions.

As discussed in detail in Section 5, our most ambitious exploration of this layer to date is the Active Bridge [3], using active extensions called *switchlets*. In this experiment, we showed that a simple switchlet-based buffered repeater could be extended (over the network) with a learning bridge switchlet and then several different spanning tree switchlets. A dynamically loaded control switchlet allowed us to perform a controlled transition from one spanning-tree protocol to another, with error detection and fall-back. This is a key demonstration, since it shows how active networking can greatly facilitate rapid changes to network protocols.

The Active Bridge is programmed using Caml, which offered a number of advantages for this application. First, like Java, Caml bytecodes are dynamically loadable and machine independent (thus permitting active extensions), and the Caml implementation is more efficient than any current Java system. More importantly, Caml gives us limited, but adequate, control over the namespace seen by the dynamically loadable modules. When combined with Caml's strong typing, this allows us considerable control over the functionality we allow active extensions to access, a feature we will refer to as *namespace security* in the rest of this paper.

A secure active router infrastructure forms the lowest layer of our architecture. While the top two layers emphasize support for several forms of dynamic flexibility, the lowest layer is primarily static. The goal of this layer is to provide a secure foundation upon which the other two layers build. The importance of this is clear, since no matter how much care we put into security of those layers, if they are loaded into an insecure environment they cannot possibly be secure.

As discussed in detail in Section 6, we embody our secure active router infrastructure as the Secure Active Network Environment (SANE). System *integrity* means that the system is not altered from some known (and presumably correct) state. SANE uses the approach of guaranteeing integrity of the lower layers of the system. While integrity is a weaker property than correctness, the importance of integrity checking cannot be overstated, as any proofs of correctness depend on the integrity of the verified components. SANE identifies a minimal set of system elements (e.g. a small area of BIOS, some cryptographic material, and a trusted source) upon which system integrity is dependent. It then builds an integrity chain with cryptographic hashes on the image of each succeeding layer in the system before passing control to that image. It also provides a public-key infrastructure that can be used for cryptographic authentication of module sources. SANE protects the assumptions about behavioral restrictions and correctness of operation that are used in minimizing the cost of per-packet operations. Thus, SANE is essential support for the 'lightweight' per-packet operation model used in designing the SwitchWare architecture.

3 A Language-Based Approach

SwitchWare provides security based on a mixture of approaches. We expect that the reader is familiar with the essentials of cryptographic-based authentication; perhaps less familiar is the programming language, specification, and formal verification-based technology we also rely upon. This section serves to present a general model of our approach to security and to explain the less familiar technologies so that their role in our model will be clear.

3.1 Security Model

There are essentially three approaches to security for facilities in active networks. We classify these as follows:

- Public facilities.
- Authenticated facilities.
- Verified facilities.

By *public* facilities we mean those that will be available to anyone, typically because the low risk of abuse does not merit the cost of restricting access to them. An example is the network service *ping*, which simply asks for an acknowledgment, and thus is a service that is often provided to the 'public'. The next level of security are *authenticated* facilities, in which a user must submit to an identity check in order to determine authorization to use a service. An example in this category is *remote login,* which typically calls for the use of a password as an access control. There is a wide spectrum of forms of authentication based on cryptographic keys that can be used to control access to such facilities. *Verified* facilities are those that go beyond the limited functionality of public facilities and the cryptographic barriers of authenticated facilities to provide facilities that are granted because of a node's ability to formally verify certain properties. This form of service has been most explored in the mobile code context, where, for instance, a type verifier can check the safety of an executing applet from an untrusted (and unauthenticated) source. An extension of these techniques would be to provide some level of authorization as part of the verification, mainly to limit the interface available to the requester, but possibly also to directly limit utilization of resources such as time, space, and bandwidth.

The technology of active networks can be viewed as a fusion of technologies from networks, operating systems, and programming languages. As such, it will integrate the safety and security techniques of each of these areas. We envision active networks as providing public facilities like the unauthenticated routing of the current Internet, authenticated facilities like logins to current operating systems, and complex verified facilities based on programming-language types and interfaces as one sees in mobile code.

3.2 Background

Before speaking in more detail of active networks, let us describe briefly the state of verification and compiler technology on which we rely.

Program Verification By the middle of the twentieth century, researchers in mathematical logic were able to describe a set of axioms and reasoning principles capable of providing a foundation for all of 'ordinary' mathematics. This foundation is mechanizable, and can therefore be implemented by computers. There was considerable optimism in the 1970's about the prospect of verifying the correctness of computer programs using logic, a task sometimes called 'program verification'. This has proved far more difficult than originally thought and it is still not considered practical to verify most large computer systems fully. On the other hand, steady progress has led to an understanding that limited properties can be be handled successfully: for instance, a compiler can construct a 'proof' that a program has a type. Modern programming languages such as Java and SML are specified in a way that allows this concept of proof to be formulated as rigorously as the logic that describes the foundations of mathematics. This precision will be useful in stating and establishing properties expected for programs written in these languages. The general trend in program verification is toward finding practical niches in which the techniques have a proper cost/benefit ratio. Areas in which this is known to be the case include hardware verification and certain kinds of system verifications involving finite state machines. Verification techniques also hold promise for mobile code, where limited safety and security properties for a class of programs need to be established with high assurance.

Type Checking Designs for strongly typed programming languages always confront a need to balance between *static* type-checking, which is done at compile time, and *dynamic* type-checking, which is done at run time. Languages such as Java and Modula-3 mainly provide static type-checking, whereas languages such as SmallTalk and Scheme provide dynamic type-checking. The choice is a trade-off between flexibility and a combination of safety and efficiency. Static checking provides greater efficiency because types are checked once at compile time. Also, with static checking errors will be detected earlier because the compiler will reject incorrectly typed programs. However, this checking must be done conservatively since it is not generally decidable whether a program will experience a run-time type error. Dynamic type-checking therefore allows greater flexibility, since this conservative restriction is avoided and type errors are dealt with at run time if they happen to occur. Many languages provide ways to balance this tradeoff by programming constructs. For instance Java permits a form of type-safe casting in which the compiler accepts the programmer's indicated type for a variable, but checks to see if the variable actually has the needed type before executing an operation on the cast variable's value.

3.3 Verification

One major advantage to using verification for active network security arises from the fact that the basic technology has been considerably developed in other contexts. In particular, one may draw on experience from operating systems, where there are well-developed techniques for using memory protection to verify at a very low level that certain memory accesses are safe, and on programming languages, where there is considerable progress on the use of type systems provide safety guarantees. The second of these is especially well illustrated in the design philosophy of Java [10], which runs on a virtual machine [14] employing a dynamic 'verifier' to enforce host security policies for the execution of the compiled bytecode of web applets. This technology is supported by advances in run-time systems (especially garbage collection) and the specification of programming languages (providing precise machine-independent semantic descriptions). However, there are two primary challenges to the application of these technologies to active networks:

1. There is a limit to the value of verification when using 'traditional' approaches.
2. Experience is needed in integrating verification with authorization.

Let us consider each of these problems in turn.

Types, as they appear in widely-used programming languages, and memory protection, as it exists in modern operating systems, are somewhat limited assurance mechanisms. While it may be important that a program does not add 'one' to 'true' or write in a particular part of memory, the specification of the problem that the program was meant to solve undoubtedly says much more than the fact that the program should not do this kind of addition or write in that part of memory. In general, programs need to satisfy a range of invariants that cannot be expressed simply. There is a need to develop techniques that can move us beyond where we are now, particularly in the areas of safety and quality of service.

An example of the kind of ideas that will be needed can be found in the *Proof Carrying Code* (PCC) work of Necula and Lee [16, 15]. PCC is based on the observation that it is often easier to check an answer than to produce it. For a mobile program, the programmer knows the key reasons it is correct (or at least safe), but not necessarily the host that receives the program. Hence it is reasonable to shift the burden of proof to the supplier of the program. The mobile program is paired with a proof of its safety and delivered to a host. It is easy for a computer to check a formal proof, even when the proof may have been very difficult to create, so the host checks the proof and runs the program. This allows program verification to deal with safety assurances in place of types or memory protection.

The second problem relates to the integration of verification techniques and authorization techniques. We require a deeper understanding of how assurances should be balanced between trust of others through cryptographic checks versus trust that is established by type checking or similar means. For instance, PCC can verify that a safety property is not violated, but it generally cannot verify that a piece of data has been correctly reported. For this, one probably needs some kind of cryptographic signature from a trusted source. Work along these lines will involve the development of various forms of 'policy languages' in which a complex set of checks involving signatures and type verifications is carried out. An obvious example is one where the interface made available to a mobile program is dependent on an identity or role associated with a signature on the program; type correctness with respect to the signature will be checked after it is known that the requester has a right to the operations in the program. Another line of inquiry involves languages for access control, such as the PolicyMaker system [6], which provides a special-purpose language for expressing policies in terms of signatures of principals and delegation of trust.

3.4 When to Check Types

Because types are a well understood and extremely effective form of mechanical verification, they play an important role in SwitchWare's safety and security model. In particular, any code that is downloaded into SwitchWare routers should be strongly typed, since this provides some basic guarantees that the router's integrity will not be compromised by the code. A question thus arises about when and where typechecking should be done. For ordinary 'immobile' code, the user often trusts (perhaps unwisely) the provider of the code, and so static compile-time type checking can be used before the program is shipped to the user. However, mobile and downloadable code like that used by active networks cannot in general be known to be trustworthy. Thus if routers are to enjoy the benefit of strong typing, they must typecheck programs themselves. Whether to do this checking dynamically or statically is different for active packets and active extensions, so we defer that discussion until later.

One other type-checking question arises. Because the code for both active packets and active extensions will be executed remotely from the programmer, making debugging challenging, it is important to make as much effort as possible to eliminate errors before the code leaves the programmer's control. In particular, even if routers dynamically typecheck the code, statically checking it before it enters the network will give the programmer a guarantee that the routers' checks will not fail. Hence, for both active packets and active extensions, we expect the programming languages we use to be strongly typed and *statically checkable*. This tension between giving static guarantees to the programmer while still requiring the routers to verify them is a general feature of our approach. This reflects our desire to improve usability by helping the programmer find errors, but improving safety and security by not trusting that the programmer was successful.

4 Active Packets

Perhaps the most radical vision of active networks is the one in which active packets (called 'capsules' in [17]) entirely replace traditional packets. Although we do not claim that this radical vision will become the dominant use of active networks, we believe it is worth exploring the idea, so that we can discover its limits. It certainly offers the ultimate in customizability and flexibility.

In SwitchWare, active packets carry programs consisting of both code and data and replace both the header and payload of conventional packets. Basic data transport can be implemented with code that takes the destination address part of its data, looks up the next hop in a routing table, and then forwards the entire packet to the next hop. At the destination, the code delivers the payload part of the data. Of course, for pragmatic reasons, our implementations actually do use some traditional headers and payloads. For example, to tunnel through the IP Internet between active routers, we encapsulate an active packet in a standard UDP packet, and transport over an Ethernet naturally requires the use of standard headers and trailers.

4.1 PLAN

We had a variety of design goals for an active packet programming language. The most important was that the language be lightweight enough that it could serve as a packet header replacement, and sufficiently well defined that we could leverage existing results in type-theory, programming language semantics, and formal methods to help us solve difficult safety and security problems. We required special remote execution facilities to capture the idea that programs transport themselves around the network by executing themselves. We required bounded resource usage, to help control denial of service attacks. For lightweight execution, we needed to provide limited enough functionality that we did not require authentication, yet we also needed to perform authorized actions when required.

Because we did not believe any existing language met or could be easily modified to meet our design goals, we have designed and implemented a new programming language, PLAN (Programming Language for Active Networks). The discussion here offers only a taste of PLAN—for more details see the PLAN overview paper [12], and our web site (`http://www.cis.upenn.edu/~switchware/PLAN.`), which includes a downloadable implementation of PLAN in Java and Caml, a language manual and user's guide, and a page where users can enter PLAN programs and execute them on our infrastructure.

```
fun ping (src:host, dst:host) : unit =
  if (not(thisHostIs(dst))) then
    OnRemote (|ping|(src, dst),
              dst, getRB(), defaultRoute)
  else
    OnRemote (|ack|(),
              src, getRB(), defaultRoute)

fun ack() : unit = print("Success")
```

Figure 3: `ping` in PLAN

Before discussing how PLAN meets its design goals, let us consider a simple example, the PLAN implementation of *ping*, as shown in Figure 3. A PLAN program consists of code, plus an entry point into that code (this provides a bit more flexibility than having a distinguished first function, such as 'main'), plus any data that make up the arguments to that initial function. For example, when `ping` is first injected into the network, the program consists of the code above, an indication that the 'ping' function should be executed, and arguments of its source and destination. When the program is evaluated, it tests to see if it has arrived at the destination. If it has not, it forwards itself on to the destination using the `OnRemote` primitive. If it has arrived, it remotely evaluates the `ack` function directly on the source, bypassing evaluation on intermediate routers, and using `defaultRoute` to guide its path.

One final point about `ping` remains: the `getRB()` calls. Each PLAN packet has a resource bound, much like IP's Time-To-Live (TTL) field, which serves to bound the total number of hops a packet and any packets it creates can take. Each hop a packet takes decrements its resource bound, and any time a PLAN program creates new packets using `OnRemote`, it must specify how much of its remaining bound will be donated to the child packets. The `getRB()` call returns the total remaining bound, and thus, in the case of `ping`, each remote evaluation carries with it all of the remaining bound.

Now consider how PLAN meets its design goals. PLAN is closely modeled on the simply typed lambda calculus with extensions to support remote evaluation. This gives it very well understood semantic and type-theoretic foundations [11], making it amenable to formal methods. PLAN only supports very simple data and control structures and thus it is easy to compile or interpret. In addition, PLAN programs can not leave behind or change state on a router and are limited in other ways that allow it to be executed without authentication. PLAN is strongly typed so that type errors cannot threaten the integrity of the router. It is statically type checkable for programmer convenience. However, since we expect PLAN programs to often only execute a small part of their code on any given router, our current implementation does dynamic type-checking at the routers. Finally, the resource bound, along with limits on CPU and memory usage on the routers, helps combat denial of service attacks.

4.2 PLANet

Recently, we have used the PLAN system to build an active internetwork, which we call *PLANet*. Based on the current Internet, PLANet currently provides a number of helpful application functions, such as reliable and unreliable datagram delivery mechanisms, as well standard network protocols, such as RIP-style routing, address resolution based on ARP, and table-based (`/etc/hosts`-style) name resolution. In PLANet, all transmitted packets are PLAN programs. Distributed protocols used to maintain the network, such as the routing and address resolution protocols, are implemented as a combination of PLAN programs and node-resident services. In particular, protocol state, timing threads, etc. are implemented on each node as services; these services communicate with their counterparts on other nodes via PLAN programs. This has the nice property that a protocol designer does not need to define new packet formats: all exchanged packets are PLAN programs, and so the packet formats are simply the standard wire representation of those programs.

The PLANet implementation runs in user-space on Linux machines and uses Ethernet as its main underlying link layer, as well as UDP as a pseudo-link layer. The implementation is written in Caml, and uses the module loader from the Active Bridge (which will be described shortly) to install new services. Despite being in user-space and bytecode-interpreted, our PLANet routers achieve about 50 Mbps for basic data delivery over 100 Mbps Ethernets. We have also run some experiments to determine the potential usefulness of active networks in improving performance, using both router extensibility and packet-level programmability. More details are available in [13].

4.3 Caml

Caml provides several of the design goals outlined for PLAN. It lacks the resource bounding (and hence avoidance of authentication) and the special remote execution facilities. However, there are many applications (such as those that require storage beyond the execution of the packet or those which modify shared state) which will require authentication in any language. In these instances, there is a simplicity gained by programming the active packets and the active extensions in a single language. In the case of remote execution facilities, Caml has provided the interface necessary to dynamically load code which has allowed us to build a general remote execution facility.

The unification of languages does have its downside. Caml was designed as a general-purpose language and we are putting it to a specific task. Our performance has suffered from the way the Caml threads package and the linker/loader operate in our domain. We continue to investigate ways in which these costs may be reduced without restricting the language.

5 Active Extensions

In SwitchWare, active packets are deliberately limited in power, making it impossible for them to be used to implement arbitrary protocols or functionality. The second layer of our architecture, the active extension layer, when combined with active packets, provides us with this power. The key insight is that this additional layer allows us to make a different set of design tradeoffs, which complement and enrich those of the other layers. For example, active packets can call extensions to provide authenticated services, making it possible to avoid default authentication for active packets. Conversely, since extensions are not mobile, they can use active packets for remote communication. The chief difference between active packets and extensions is that although extensions may be (but are not required to be) dynamically loaded across the network as active extensions, they execute entirely on a particular node. Thus extensions are base functionality or are dynamic additions rather than 'mobile code'.

In general, we expect that active extensions will be loaded onto routers, and will then provide services to many active packets. Thus, it is reasonable to subject active extensions to heavier-weight security checks than active packets. Thus, we expect that active extensions will be statically type-checked upon arrival on a router, and it may be that some will also carry credentials with them, such as cryptographic signatures and proofs of correctness. Checking these credentials may be expensive, but will need to be done only once. Because of this heavier-weight checking, active extensions can be allowed access to facilities in the router that active packets cannot. In particular, creating or changing state on a router, or direct access to the routers network interfaces, must be done by extensions. Interestingly, this implies that the PLAN interpreter itself is an extension, albeit probably not one that is dynamically loaded.

Before moving on to a case study that shows some of the power of switchlets, it is important to clarify a bit of terminology. In PLAN, we refer to the 'service' level as the part of the system that PLAN can call to perform actions denied to PLAN itself. Extensions are used to implement PLAN services. However, active extensions are more general than PLAN service implementations, and, in fact, they are not required to provide PLAN-accessible interfaces (although they must do so if PLAN active packets are to invoke them directly).

5.1 The Active Bridge: A Case Study

The Active Bridge [3] is a prototype constructed to study active networking at the active extension layer. It is used to bridge 100 Mbps Ethernet LANs, with the additional 'active' feature of automated recovery from failure of an implementation of a spanning-tree algorithm, achieved with the use of active extensions called *switchlets*. The current system is written in Caml, and runs on the Linux and OpenBSD operating systems, and we have deployed it on Intel architecture machines. A high-level architecture of the system is shown in Figure 4.

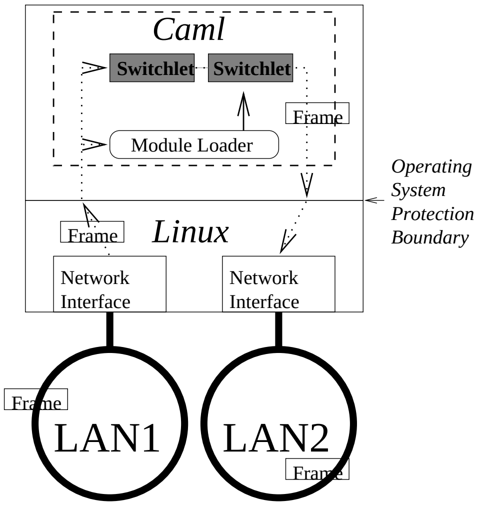

Figure 4: The Active Bridge Architecture

The Active Bridge is based on a module loader (the Active Loader) and a set of core system services. The core services include access to external services such as raw Ethernet frame I/O via Linux sockets. A switchlet is loaded into the system to provide the functionality of a buffered repeater, and therefore provide simple LAN interconnection. Since the aggregate performance (throughput) of such a system is limited by the line rate of the LANs it interconnects, a self-learning functionality is added with an additional switchlet; this allows frames which need not be forwarded to be dropped, preserving bandwidth on other LANs and obtaining any gain possible from network locality.

Since loops can exist in bridged networks, resulting in the endless circulation of Ethernet frames, a spanning tree algorithm (STA) is used to define topological restrictions on the frame forwarding at each node. This spanning tree algorithm is loaded, as before, as a switchlet. In fact, two are loaded: one for the DEC STA and one for the IEEE 802.1D standard. One of them is, by design, in error. A final switchlet performs sanity checks on the generated spanning tree; when a frame of either DEC or IEEE type causes a transition in the STA, the result of the tree construction is checked; if an error has occurred the STA control switchlet restores the previous STA to the execution path.

The previous discussion illustrates an important difference between the SwitchWare architecture and other proposals such as MIT's capsules. The capsules model is focused on per-packet execution; the follow-on work in the Active Network Transport System (ANTS) loads functions as necessary to allow packets to execute. The BBN Smart Packets model [5] provides a very dense CISC-like language that is biased towards management tasks, which assumes that nodes are essentially stateless. The SwitchWare architecture, by contrast, *layers* packet execution and service loading. PLAN provides the user-access model, while the

Active Bridge core services and module loader provide the basis for controls such as levels of privilege, priority, resource scheduling, and so on.

With the Active Bridge, we have done a set of experiments to understand both the costs of loadable modules and the cost of our demultiplexing architecture. To test the first case, we have a version of the Active Loader which supports only the Bridge. In this case, we see a throughput as measured by `ttcp` of 61 Mbps. Our second experiment used the general Active Loader and gave a throughput of 54 Mbps. Of the per-packet processing time, approximately 70 μs is calling `recvfrom()`, 70 μs is processing in the Bridge, and 30-40 μs is calling `sendto()`.

5.2 Namespace Security

For SwitchWare, an essential feature of the Active Bridge is its model of module loading, which provides a flexible form of namespace security. The idea depends on the fact that Caml, like many other languages, provides access to functionality through explicitly specified interfaces. The restrictions imposed by strong typing make it impossible to access functions, data, types, and so on except through the names provided by these interfaces. Analogous ideas arise in object-oriented programming through restrictions based on methods, which also appear as names. However, Caml and other ML variants go one step beyond standard interface protection schemes by supporting a form of restriction known as module interface *thinning*. The idea is that a module interface may provide access to functionality that is needed by some parts of the system, but to which some other modules, especially dynamically-loaded ones, should not have direct access. For example, the built-in error logger in the Active Bridge needs to be able to write to disk, but dynamically-loaded modules should only be allowed to do so indirectly by using the error logger. The dynamic loader provides this capability by allowing a module to be 'thinned', which means that names are removed from its interface, before a new module is dynamically linked. If the new module tries to access a name that is not present (either because it was never present, or because it was thinned) then the dynamic loading process fails, and the module is rejected.

One interesting area for further work is to combine this language-oriented namespace protection with cryptographic protection. For example, some interfaces could be provided only to active extensions that could prove cryptographically that they came from the router vendor and were being installed by the router owner. A general theory based on 'who you are determines what you see' could be derived from an effective combination of interface thinning for modules and policy languages for authorization.

6 Secure Active Routers

As can be seen from the previous sections, the SwitchWare architecture depends, at least in part, on language systems to guarantee the safety and security of the active network, as active packets arrive, act, and depart from a node. While it is seductive to trust other portions of the architecture to perform as we expect, we believe that it is crucial to explore and analyze the requirements for building a secure active router infrastructure as part of the SwitchWare architecture. This infrastructure is the solid base upon which active packets

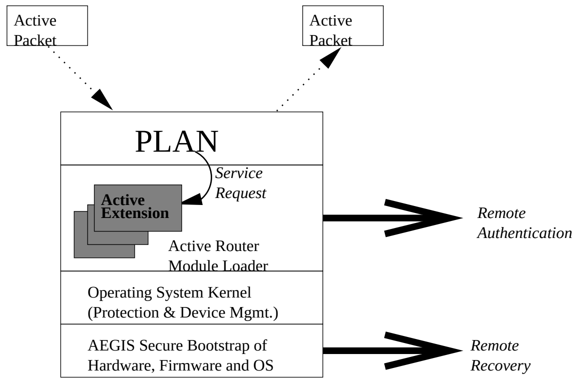

Figure 5: SANE's Relation to SwitchWare

and active extensions build. The goals for our secure active router infrastructure include the following:

- To support the language-oriented model used at higher layers of the SwitchWare architecture;
- To incur minimal costs while the system is an operational state, by migrating costs to an infrequently performed pre-operational phase; and
- To maximize system security under a minimal set of assumptions about trusted components.

The language-based schemes offer the benefit of provable behavior, and hence an opportunity for building provably secure systems. For the proofs to succeed, however, their assumptions about the 'state of the world' must be true. The secure active router infrastructure guarantees that these assumptions (e.g. that a certain version of system microcode is installed) are true. Thus the security of the SwitchWare architecture as a whole is grounded in this layer.

6.1 SANE

To embody our secure active router infrastructure, we have designed the Secure Active Network Environment (SANE) [1]. SANE provides an architecture with: a demonstrably minimal set of trust assumptions, the ability to securely bootstrap the remainder of the system when the trust assumptions are met [4], and authentication and naming services for code that is loaded. In the context of the SwitchWare architecture, the role of SANE is to ensure that the presumptions of the other system elements, such as the active packet or extension layers, are true. An illustration of the SANE in the context of the overall SwitchWare network element architecture is shown in Figure 5.

The SANE elements are combined into a system using the following design principles:

- Dynamic checks, performed while the system is operating, should be as fast as possible, as they are done many times;
- Static checks, performed before the system enters the operating state, can be more expensive, as they are done only once;
- System performance can be improved by tradeoffs that decrease the cost of the dynamic checks at the expense of more costly static checks, or ideally by using static

checks to eliminate the need for any dynamic checking at all (analogous to once at compile time versus many at run-time).

We have done an initial implementation of SANE and made some preliminary measurements to help us understand the inherent costs of our active environment and the overhead imposed by its security services. We wrote an active version of `ping`, which achieves a round trip time of 5 ms when unauthenticated. The authenticated version's round trip time is 8 ms. If we factor out the 990 μs taken by the kernel and the transmission, about 70% of the time in the unauthenticated case is in linking the active ping code to our environment, with 40% spent in updating the symbol table specifically. In additions to these costs, the authenticated version spends 3 ms for the 4 cryptographic keyed-hash calculations (one Message Authentication Code calculations and verification in either direction).

For more details on SANE, see [1].

7 Related and Future Work

This volume is an excellant overview of related work on active networks. A comprehensive survey of active network research as of about one year ago [17] delineates some of the key differences in approach and research directions of the early efforts. The papers on PLAN [12], the Active Bridge [3], and SANE [1] all contain extensive comparisons with related efforts.

A great deal of exciting work lies in front of us. While active networks might be viewed as revolutionary, evolutionary (or even devolutionary!), we view our work in the SwitchWare effort to be one of evolving our approach under experimental scrutiny. While the elements of the system are piecewise implemented, the software interfaces provided by each layer of the system will change as experiments demonstrate strengths and weaknesses of various approaches. Since many of our design assumptions about function placement depend on the nature of packet flows, and therefore on the nature of active applications, we are developing applications that can be used to test and evaluate our infrastructure as well as other proposals. Among the more interesting applications are flexible architectures for multicast, an example of which we described in the introduction, and network infrastructure support for *booster protocols* [9], a design methodology that adds protocol functions as-needed, and is thus ideal for deployment on an active network infrastructure.

A second major thrust is interoperability with other active networks research efforts. While several of the earliest active networks efforts selected functions on the basis of a matrix of required contributions to make the effort as a whole complete, there has been considerable blurring of these roles and contributions as the work has evolved. One effort we have pursued with other members of the active network community is the design of an 'Active Network Encapsulation Protocol' (ANEP); this has grown into perhaps the first active network RFC [2]. ANEP has been adopted and used in many of the active networks projects, and has allowed packet forwarding across the IP Internet. To achieve greater interoperability, however, we must provide more than a standard packet format; this, after all, is the essence of the Internet idea. What we must provide is a means of forwarding flows and functions among and across a variety of active network node types, so that user applications can be written that are oblivious to many particulars of the programmable network infrastructure they are traversing or using. Effective integration for SwitchWare might mean, for instance, that the ANTS transfer mechanism [18] could provide a good distribution mechanism for SwitchWare extensions, or the micro-protocols of the Ensemble Project [8] could provide a rich set of SwitchWare services that can be configured by PLAN packets.

8 Conclusions

The SwitchWare active network architecture integrates the necessary components of any active network element. The integration takes the form of a layered architecture, with functions partitioned between layers based on the flexibility and security tradeoffs required at each layer. Higher levels of the system provide more restricted functionality, with one consequence being that they provoke commensurately less security risk. A second important consequence of this layering is that the higher levels can operate with 'lightweight' checks on their behavior. Since we believe that the common case will be handled by our highest layers, such as PLAN, SwitchWare provides an attractive point in the space of security, flexibility and performance tradeoffs.

We believe that such an integrated architecture represents the key to any verification or validation of an active network element. The language-based approach we described has thematically unified each of the constituents of SwitchWare: PLAN, the dynamic module loading infrastructure, and SANE. Since each of the languages or language systems involved represents an abstraction that can be used as part of a system verification process, we believe that the approach offers the a good path to a verified network composed of active network elements.

Much of the software described here is available over the Web. The SwitchWare homepage is

`http://www.cis.upenn.edu/~switchware`.

It provides a starting point for locating the software and further documentation, including many of the documents we have referenced and links to most related active network projects.

Acknowledgments

Hicks, Kakkar, and Moore implemented PLAN and PLANet; they, together with Gunter and Nettles, are responsible for their design as well. Alexander and Marianne Shaw designed and implemented the Active Bridge. Arbaugh designed and implemented AEGIS, the secure bootstrap method used in SANE. Arbaugh and Keromytis designed AEGIS recovery. Alexander, Arbaugh and Keromytis architected SANE. We thank David Farber of Penn, and Dave Sincoskie, Bill Marcus and Mark Segal of Bellcore for many valuable technical discussions. This work was supported by DARPA under Contract #N66001-96-C-852, with additional support from the Intel Corporation.

References

[1] D. S. Alexander, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. A Secure Active Network Environment Architecture, 1998. This volume.

[2] D. S. Alexander, B. Braden, C. A. Gunter, A. W. Jackson, A. D. Keromytis, G. J. Minden, and D. Wetherall. ANEP: Active Network Encapsulation Protocol. `www.cis.upenn.edu/ switchware/ANEP`.

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

[4] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. A Secure and Reliable Bootstrap Architecture. In *IEEE Security and Privacy Conference*, pages 65–71, May 1997.

[5] Smart packets. `http://www.net-tech.bbn.com/ smtpkts/ smtpkts-index.html`.

[6] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[7] Caml home page. `http://pauillac.inria.fr/ caml/ index- eng.html`.

[8] Ensemble home page. `http://simon.cs.cornell.edu/Info /Projects/Ensemble`.

[9] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. S. Bakin, W. S. Marcus, and T. M. Raleigh. Protocol Boosters. *IEEE Journal on Selected Areas in Communications, Special Issue on Protocol Architectures for the 21st Century*, 1998.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[11] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. The MIT Press, 1992.

[12] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Programming Language for Active Networks. `http://www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`, 1998.

[13] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An Active Network Testbed. `http://www.cis.upenn.edu/~switchware/ papers/planet.ps`, 1998.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[15] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM Press, 1997.

[16] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Second Symposium on Operating System Design and Implementation (OSDI '96)*, 1996.

[17] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[18] David Wetherall, Ulana Legedza, and John Guttag. Introducing new internet services: Why and how. This volume.