PLAN System Security

Michael Hicks

July 14, 1998

1 Introduction

Active Networks offer the ability to program the network on a per-router, per-user, or even per-packet basis. Unfortunately, this added programmability compromises the security of the system by allowing a wider range of potential attacks. Any feasible Active Network architecture therefore requires strong security guarantees. Of course, we would like these guarantees to come at the lowest possible price to the flexibility, performance, and usability of the system.

The PLAN system is a distributed programming framework we have used to build an Active Network, *PLANet* [4]. In the PLAN system, code implementing distributed programs is broken into two parts: the *PLAN level*, and the *Service Level*. All programs in the PLAN level reside in the messages, or packets, that are sent between the nodes of the system. These programs are written in the Programming Language for Active Networks [6] (or simply, PLAN). PLAN programs serve to 'glue' together Service level programs; PLAN may be thought of as a network scripting language. In constrast, Service level programs (or simply, services), reside at each node and are invoked by executing PLAN programs. Services are written in general-purpose languages (in particular, the language that the PLAN interpreter is written in) and may be dynamically loaded.

The current Internet (IP and its supporting protocols) allows any user with a network connection to have some basic services. In addition to basic packet delivery provided by IP, basic information services like DNS, `finger`, and `whois`, and protocols like HTTP, FTP, TCP, SMTP, and so forth are provided. Similarly, a goal of PLANet is to allow any user of the network to have access to basic services; these services should naturally include some 'activeness.' This goal implies that some functionality, like packet delivery in the current Internet, should not require authentication; in PLANet, *we allow all 'pure' PLAN programs to run unauthenticated.* A PLAN program is considered 'pure' if it only makes calls to services considered safe; for example, determining the name of the current host is a safe operation, while updating the host's router table is not. Successfully calling unsafe services would require proper authorization. This security policy is stated more formally in the following subsection.

1.1 Security Architecture

For the current PLAN system we propose (and have partially implemented) a *hierarchical* security infrastructure in which privilege can be viewed as a poset, as shown in Figure 1. Here, $\bot$ represents privilege allowed to all unauthenticated PLAN programs, while $\top$ represents privilege allowed to the node administrator. Intermediate nodes represent varying levels of privilege such that a principal with privilege level m may additionally invoke services with levels $\leq m$. Privilege is checked and enforced when a service routine is called from a running PLAN program.

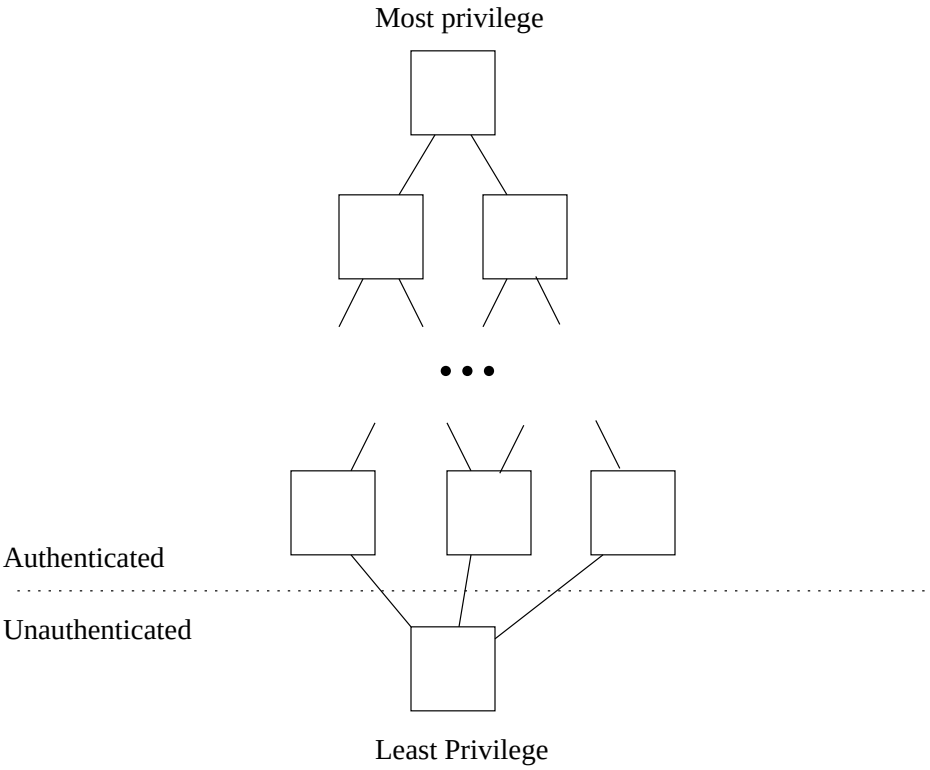

Figure 1: Security Architecture as a Poset

There are important ramifications of this architecture. The privilege provided to any particular set of principals at level m (represented by a node in the poset) draws on the privilege the children of m. In particular, a breach in security at any point in the poset affects all all parents of that point. Therefore, *the foundation of our security policy is the safety and security of unauthenticated PLAN programs.*

This paper presents a series of additions and changes to the PLAN system to implement this security architecture. In particular, we desire to

1. **strengthen the security of PLAN programs**, the foundation of the architecture. We do this by
   a. further restricting the PLAN language to a 'safer' subset
   b. allowing node policy-specified bounds on elapsed time and memory consumable by a PLAN program.
2. **add trust management services to manage security at the higher layers**. In particular, we make use of the Query Certificate Manager [9] to add mechanisms to PLAN to
   a. describe the sets of principals associated with the nodes in the poset
   b. describe and enforce privilege allowed each set of principals

We first present the perceived threat model to PLANet followed by the motivation for the particular solutions we present. We then fully describe the design and implementation of our changes, including performance analysis.

2 Threat Model

The two major threats to the PLAN system are to the *public resources* of the system: the CPU, memory, and network; and to the *contents* of the system: the packets themselves and the information stored on routers. The more specific forms these threats might take are outlined below.

- **Denial of Service**. This boils down to resource control. Because of the potentially resource-intensive nature of active programs, the resource control model of an active node must be different from a traditional passive one. We want to strictly control:
  - *CPU time.* This is a question of program execution time and proper scheduling. All programs should only have a limited number of CPU cycles (they must terminate), and each program should have "fair" access to the node CPU. In particular, we would like those programs which require minimal servicing to have priority over those which are more computationally intense: "dumb" packets which require only a routing service should preferentially processed over all others. Furthermore, when the node is processing some "active" packet, it should not deny service to other packets which arrive in the meantime. We might also like to provide a more liberal environment to authorized users.
  - *Memory.* While a program executes, it occupies memory which is released upon termination, termed *ephemeral state.* If a program is allowed to occupy too much memory, even for a finite period, service could be denied to other programs. Therefore memory allocation must be controlled. One policy might be to allow a program only to allocate an amount of memory proportionate to its own size.

Furthermore, an active program might reserve memory on the node beyond the program's own execution, termed *resident state*. Such state may reside on the node until explicitly removed (*persistent state*), or it may last only a finite period (*temporary state*). As examples, resident state could be the result of adding entries to a routing table, installing new services on a node, or marking nodes as part of DFS. We would likely want newly installed services to persist, while application-specific data would be temporary (such as node marking for a network DFS). The allocation of resident state must be strictly controlled, probably via authentication, if memory resources are to be available to all packets.

The *worth* of memory allocated is tied to the running time of a program. For example, if two programs P and Q allocate n bytes of memory, but P terminates quickly while Q retains its memory for a much longer period (say, because it is blocked on a synchronization primitive), then Q will have a greater system impact. This is because it is more likely that other programs will arrive while Q is still running and thus contend for that memory. It may therefore be desirable to have a memory control policy which is tied to CPU time.

- **Snooping**. This boils down to isolation and authorization. In particular, one program should not be able to look at data private to another program unless it has been authorized to do so. This data could be resident state, left by a previously running program, or it could be state from a concurrently-running program. It should also be possible to protect packets from snoopers while in transit.
- **Spoofing**. Some desirable security policies (such as the control of memory and CPU, or access to sensitive information) require that the packet associate itself with a particular principal so that privilege due that principal can be enabled. Therefore, it is particularly important that the identity of the principal cannot be spoofed. This is also important for distributed protocols, such as routing protocols: information claiming to be from a particular node/service should be authentic.

3 PLAN

For PLAN programs to run unrestricted as the foundation of our architecture, there must be some properties of the language itself that prevent attack; that way all programs written in PLAN are 'safe.' The language is resource- and expression-limited, thus preventing certain types of denial of service and safety-compromising attacks. For example, all PLAN programs are guaranteed to terminate[1], since PLAN does not provide a means to express non-fixed-length iteration or recursion. Additionally, PLAN programs are isolated from one another, since the language's strong typing prevents pointer-swizzling or other attempts to get at the underlying representation of objects, and because there is no means of direct communication among PLAN programs.

While PLAN language restrictions can bound execution on a single node, they are not sufficient in restricting use of network resources. Consider the following program[2]:

```
fun ping_pong(pingHost:host, pongHost:host) : unit =
  OnRemote (|ping_pong (pongHost, pingHost)|,
            pongHost, getRB (), defaultRoute)
```

Unchecked, this program will jump back and forth between two nodes forever. To alleviate this problem, PLAN packets have a *resource bound* counter which is decremented each

[1] PLAN programs terminate as long as the services called also terminate.

[2] A basic knowledge of PLAN is assumed. See [5] for a detailed description of programming with PLAN.

time a new packet is sent. Therefore, the number of hops that a PLAN program or any of its progeny may take are limited by the initial value of this counter. This mimics the functionality of the IP TTL field.

However, PLAN alone, even with the resource bound counter, is not adequate. For instance, while PLAN programs will terminate *eventually*, nothing in the language guarantees *when*. Consider the following program:

```
fun f1():unit = ()
fun f2():unit = (f1(); f1())
fun f3():unit = (f2(); f2())
fun f4():unit = (f3(); f3())

fun exponential():unit =
     (f4(); f4())
```

This program (beginning at the `exponential` function) executes an exponential number of function calls compared to the number of function definitions. Furthermore, memory can be allocated at an exponential rate as well:

```
fun exponential():unit =
  let val a:int list = [1;2;3]
      val b:int list = a @ a
      val c:int list = b @ b
      val d:int list = c @ c in
    ()
  end
```

Each call to the *append* function (written here as `@`) will essentially copy the current list and then append the two lists together. `@` can be written in PLAN, though it is not shown here.

Unrestricted exponential allocation and execution are clearly unacceptable. The question is, what is a more acceptable policy? Looking to the current Internet, we see the resources required to process a packet are linear in the size of the packet. In particular, in a store-and-forward router, the router must allocate enough space to queue the packet for processing; this requires time and space in proportion to the size of the packet. On a cut-through router, only the header is stored while the packet itself is 'cut-through' the router based on the forwarding decision; this is a constant time operation. The forwarding process itself is (in general) constant in relation to the packet.[3]

To force PLAN programs to execute in time linear to their size, we could do a number of things:

- **Further restrict PLAN.** For instance, we could limit all functions in a PLAN program to calling at most one other PLAN function. This would prevent the exponential growth we saw earlier, and based on our current experience, would not be overly restrictive to PLAN programmers. Furthermore, compliance could be checked statically by a simple compile-time flow analysis. Of course, the drawback is decreased programming convenience.

- **Use scheduling tricks.** Programs can run for a long time, but they obtain the CPU exponentially less often (when there is contention for it). This can be implemented using a priority scheduler with an aging policy. The problem is that this turns the termination question back on the user: rather than the router being unconvinced

[3]This is for the general case, not including features such as source-routing, fragmentation, options processing, etc.

about *when* a program will terminate, now the user has the same problem. As time passes, this scheme resembles CPU limits (see below) enough that it may not be worth the added complexity of implementing it.

- **Add CPU and memory allocation limits.** This has the advantage that packet execution and allocation is *constant* in relation to the size of the packet, as in the cut-through router case. The major drawback is that limits remove any correctness guarantee from the PLAN program, since it may terminate adversely without completing its task. Of course, such a guarantee is already absent thanks to the resource bound counter, and the fact that remote evaluation is unreliable. However, this counter-based uncertainty is more easily managed and understood by the programmer, and is also architecture-independent. Implementing CPU and memory limits efficiently is also not straightforward.

One question still remains: is a linear resource bound enough? In the case of IP routers, the coefficient is likely to be very small: just the copying cost. With a PLAN program, this constant is likely to be much higher. In some sense, the termination question comes up again: is it enough to know the asymptotic bound on resource usage, or is a tighter bound required? We believe that to answer this question requires experimentation.

In the next subsection we present one possible PLAN restriction. In the following subsection we present the implementation and performance of CPU and memory restrictions.

3.1 PLAN Language Restrictions

Semantically speaking, language restrictions would be the most ideal solution to the problem of resource usage since program correctness could be preserved. The difficulty is in formulating restrictions that are adequate, that do not overly limit expressiveness, and are not too costly to enforce. We propose one such restriction here, but acknowledge more work in this area is needed.

Execution and allocation exponential in the length of the program can be attacked by further restricting what constitutes a legal PLAN program.

1. Limit all PLAN functions as follows:

   Given function f which calls functions $g_1,\ g_2,\ \ldots\ g_n$:

   $$\begin{aligned} f \quad \in \quad & \mathsf{valid}\ \textbf{iff}\ g_1,\ g_2,\ \ldots\ g_n \in \mathsf{valid}\ \textbf{and} \\ & \mathsf{calls}(f) = 0\ \textit{or} \\ & \mathsf{calls}(g_1) + \mathsf{calls}(g_2) + \ldots + \mathsf{calls}(g_n) \leq 1 \end{aligned}$$

   where $\mathsf{calls}(g)$ is the number of PLAN functions called from function g.

   This could be checked at runtime with a simple counter initialized at function entry, and a compile-time flow analysis could be used to check for *a priori* compliance.

2. Make it so that each call to the function given to *fold* subtracts 1 resource bound. This can be thought of as a call to `OnRemote` for each iteration. In fact, we might extend this analogy further: allow recursive function calls in PLAN such that each recursive call subtracts one resource bound. Given either restriction, writing a function like *append* will subtract one resource bound for each allocation. This prevents exponential allocation.

Each of these restrictions assume that no (unauthenticated) service call will execute in time exponential to its input, or worse, return an output that is exponential in its input size.

Table 1: Summary of Changes to Thread Scheduler

Thread datastructure extended to contain

- Memory consumed (updated on CS and query)
- CPU time consumed (updated on CS)
- Cleanup handlers invoked on thread death

Thread interface extended:

$$\begin{aligned} push\_handler &: (unit \to unit) \to unit \\ pop\_handler &: unit \to (unit \to unit) \\ running\_time &: t \to float \\ alloc\_words &: t \to int \end{aligned}$$

3.2 CPU and Memory Limits

In this subsection, we describe the implementation of imposing CPU and memory limits on PLAN programs. The required changes were all to the OCaml thread scheduler, and are summarized in Table 1.

PLANet implementation

A few details about the current PLANet implementation are in order. PLANet is implemented in OCaml, a byte-code interpreted dialect of ML. Services may be dynamically loaded into running nodes by delivering (via PLAN packets) and installing OCaml byte-code. Our testbed consists of 300 MHz Pentium-II's connected by Fast Ethernet, running Redhat Linux, kernel version 2.0.30.

The implementation makes use of threads. While POSIX-based threads are available, the implementation is buggy; we instead use the user-level threads package provided by the distribution. Threads are preemptive, and the quantum is set to 50 ms. Unfortunately, the scheduler is not well-tuned, and so context switch times are rather high—around 110μs. It turns out that the threads system is a major bottleneck in our performance.

OCaml is a garbage collected language. OCaml uses a generational collector in which each thread has its own allocation area which is copy-collected into a single, shared major heap. The major heap is managed by an incremental mark-and-sweep collector. This fact allows us to (relatively) easily track per-thread allocation.

CPU limits

Once a PLAN program has executed past its allowed limit, it must be killed. This implies that the PLAN interpreter must be able to:

1. **Track the time spent executing a given PLAN program.**
   If multiple PLAN programs may execute concurrently, each one should only be charged for the time it has the CPU, not while it is in the scheduling queue or blocked on I/O.
2. **Halt the program execution, even while in a service call.**
   This implies that it is not enough to have instruction counters within the interpreter—services themselves must be interrupted to halt execution.

The first requirement implies that we need to obtain information from the OCaml thread scheduler about per-thread execution times. Therefore, we have extended the information stored by the scheduler about each thread to include the amount of CPU time used thus far. This value is updated each time a context-switch occurs and is made available in the threads package interface.

The second requirement implies that a thread running PLAN programs can be killed safely. The issue here is that the PLAN program could be in the middle of a service call which is executing within a critical section. If the program is killed, then mutex guarding the section is never released, preventing other programs from accessing the data. Furthermore, it is not enough to simply keep track of mutex variables and release those at thread-death. The reason is that a critical section presumably guards a *transaction* in which all or none of the operations should take place to preserve consistency.

The approach that we take resembles that taken by POSIX threads. With each thread we associate a stack of *cleanup handlers*. When the thread is killed, each handler is popped off and executed. Using this mechanism, we can code critical sections as follows. We alter the call to lock a mutex to also push a handler to unlock that mutex. If the thread is killed during the critical section, the mutex will be unlocked by the cleanup handler. If the critical section completes normally then rather than calling unlock directly the cleanup handler is popped off of the stack and invoked. By allowing only critical sections to be coded we can be sure that the stack semantics of the cleanup handlers will be correct.

Another possibly more elegant solution would be to cause a thread to asynchronously raise an exception when it is killed. The exception could then be caught by contextual code to perform cleanup. This is the approach taken by Java. The difficulty here is that the death of the thread depends on the exception reaching the top-level, which can be prevented by catching and not rethrowing the exception.

Another approach that we've thought about to make thread-killing safe is to eliminate blocking synchronization entirely, in favor of non-blocking synchronization (NBS) [2]. A convenient atomic operation needed by NBS algorithms is DCAS ("Double Compare and Swap"), which we believe can be safely coded in the context of the OCaml threads system. More research is needed in this area.

Finally, we must decide when and how to halt a misbehaving program. The question of when is decided by node policy – currently we allow the specification of both an elapsed time (which counts time spent blocked in a service or on I/O) and a CPU time limit. Enforcement is done via a 'watchdog' thread which runs in parallel with executing PLAN program. The watchdog thread wakes up in short increments and polls the current memory (see below), CPU, and elapsed time statistics of the thread evaluating the PLAN program. Once that thread surpasses a specified threshhold, the thread is killed.

Memory limits

Just as it is not enough to have a PLAN instruction counter to track execution time, it is not enough to have a PLAN allocation counter; this is because memory may be allocated as a result of a service invocation. Therefore, we rely on the OCaml runtime system to track systemwide allocation. The runtime currently tracks garbage collection and memory allocation information on a system-wide, rather than per-thread basis. Therefore, we have extended the information stored by the scheduler about each thread to include the amount of memory allocated thus far. This value is updated each time a context-switch occurs and is made available in the threads package interface. We additionally update the counter whenever its current value is queried; this enables a thread to check its own allocation as it goes.

As indicated above, the watchdog thread polls the memory counter at regular increments

and compares it against the allowable threshhold, killing the PLAN thread if necessary. Since PLAN programs can allocate exponentially (as we saw earlier), there is the worry that the threshhold will be non-trivially exceeded before the watchdog checks the counter. We haven't found this to be a problem in practice.

One problem with this approach is that the kind of memory allocated by the node due to a particular PLAN program is not classified in any way. This prevents the node from making use of classification-based policies. For example, we might want to say that a PLAN program can allocate n bytes of *temporary state* (GC'ed when the PLAN program finishes), and m bytes of *resident state* (state that remains when the PLAN program halts). We might also like to make further sub-classifications, such as: all resident state allocated by service X should not be attributed to the PLAN program (since the service presumably manages its own state).

Performance

We ran some performance tests to determine the overhead imposed by these changes. We first ran some microbenchmarks to determine basic overheads. We found that, as expected:

- The basic cost of a thread context switch was increased from 110 to about 115 μs since it has the additional task of updating CPU and memory information.
- The cost of a entering and leaving a critical section was higher, up to 7 μs from 3 μs, since the appropriate handlers must be pushed on and popped off of the thread handler stack.

To understand the effect of these additional overheads, consider that the per-packet processing time of our router is currently about 240 μs, with 150 μs being spent in the code itself (the rest is due to kernel crossings). During this time, we enter five critical sections, so these overheads are increased by 20 μs. In addition, we can expect that a single thread of execution from packet arrival to packet delivery will only be affected by a context switch every 294 packets, so the minimal addition to the context switch overhead should be negligible. If these were the only overheads, we would expect our switching performance to drop by about 8%. However, as Figures 2 and 3 show, performance was degraded significantly more than this. Both figures compare performance with hop count, where one hop away signifies the machines are on the same LAN, and two hops means that the packets are routed. Numbers for versions of PLANet are shown with the additional overhead of the timing mechanisms appearing in white.

Figure 2 illustrates the latency of performing an 'active ping' with 0 and 1437 byte payloads. The white part of each bar is the additional overhead incurred by adding the accounting mechansims. We can see that the difference in elapsed time for each case is nearly constant – about 0.3 seconds. This has the effect of degrading latency by between 11 and 18% from the original version. More striking is the contrast shown in Figure 3: 46% less bandwidth for nodes on the same network (1 hop), and 30% less bandwidth for the routed traffic (2 hops). Note here that the white part is not the overhead but the total reduced bandwidth. In both cases, the bandwidth is limited by the receiver. This is because this is where evaluation of PLAN packets occurs, which is now extremely costly, as explained below.

The measured loss in performance is much worse than our predicted 8%. This is due to the overhead in the OCaml user-level thread scheduler. As mentioned, context switch times are upwards of 110 μs, which means we want to minimize them as much as possible. In the regular implementation, we do this by making packet-processing single-threaded: a single thread watches all of the network devices, grabs a packet, evaluates the PLAN program within (or routes it), and then looks for the next packet. No other threads run

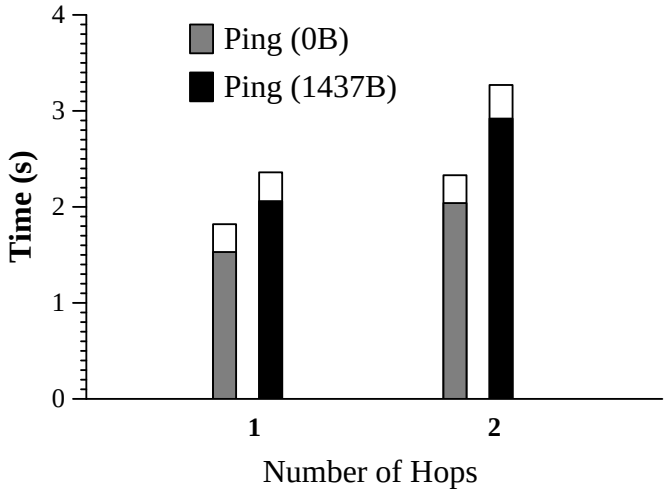

Figure 2: Effect of Resource Bounds on Latency

very frequently, and so this 'worker-thread' is not bothered by excessive context switches. However, when packets are to be evaluated (which occurs in all of our experiments only on the endpoints), we must additional make use of the watchdog thread to guard resource use. In order to do so effectively, the worker thread now has to fork two additional threads for each packet to be evaluated – one to evaluate the packet, and one to be the watchdog thread. The worker thread then waits for the evaluation thread to complete (which could happen before the watchdog thread wakes up for the first time at least 50 ms later), and then continues. This means that at least one context switch will occur per packet, which nearly doubles the time to process a packet. There is also the additional cost of thread creation and cleanup added per-packet.

In general, our feeling is that better runtime system support is needed to implement resource control policies efficiently. We'd like a threads system with small context-switch times and the ability to associate resource policies (covering memory, CPU, disk, etc.) with a thread which can be efficiently enforced, either by the system or by the program. Ideally, these threads would map to OS-level threads to take advantage of multiple processors (not currently available in OCaml). A QoS operating system such as Nemesis [7] might be a reasonable basis for such a system. Alternatively, a more suitable implementation language, such as Erlang [1], might be used.

4 Trust Management

Privilege to invoke potentially unsafe services from PLAN programs is described and enforced by a system of trust management. In particular, our architecture makes use of sets of principals, where each principal is identified by a public key, such that each member of the set is granted the same privilege. The sets are set up so that they are non-overlapping: more privileged users expand the trust alloted to their less-privileged counterparts. When a running PLAN program invokes a service which requires privilege, the principal associated with the packet is authenticated, and then the operation is authorized. If either authentication or authorization fails, the operation is not permitted.

In this section, we describe mechanisms used by PLAN programs for authentication and authorization. For the former, we choose to take an approach in which service calls

Figure 3: Effect of Resource Bounds on Bandwidth

requiring privilege must be executed in the context of some authenticated code. For the latter, we use the Query Certificate Manager [9] to describe sets of principals as well as their privilege relationships. These two approaches are developed and described in the next two subsections.

4.1 Authentication

Before a PLAN program may invoke a trusted service, its associated principal must be determined; this is the process of authentication. Authentication is typically done in a public-key setting by verifying a digital signature over some piece of data. The first question, then, is what piece of data will be signed? One thing that suggests itself is a PLAN *chunk*.

A chunk (or **c**ode **hunk**) may be thought of as a function that is waiting to be applied. In PLAN, chunks are first-class, and consist internally of some PLAN code (if necessary), a function name, and a list of values to be used as arguments during the application. Chunks are specified syntactically in PLAN by surrounding a function call with |'s.. A chunk is typically used as an argument to `OnRemote` or `OnNeighbor` to specify some code to evaluate remotely. A chunk may also be evaluated locally by passing it to the `eval` service, which resolves the function name with the current environment, performs the application, and returns the result.

We've added an additional service called `authEval` which takes as arguments a chunk, a digital signature, and a public key, the latter two represented as PLAN values of type `blob` (meaning unstructured data). `authEval` verifies the signature against the binary representation of the chunk, as defined by the standard PLAN program wire representation. If successful, the chunk is evaluated as in `eval`; otherwise, the exception `AuthenticationFailed` is raised. During the evaluation of the chunk, the authentication code keeps track of the authenticated principal by associating it with the current thread id. When the chunk terminates, the association is removed. This way, if any service invoked as a result of chunk evaluation wishes to perform authorization, it may ask the authentication service for the principal. Because a caller's thread id cannot be forged, this provides a safe way to track a principal without worry that some malicious service will change the associated principal after the authentication phase.

There are two key advantages to this approach. One is that a principal signs exactly

the piece of code he wants to execute, and may only have extra privilege while executing that piece of code. Secondly, only those programs which require authorization will have the extra time and space overheads.

There are also three problems with this approach. The first is that the authentication performed here is *one-way authentication*. While the program is authenticating itself to the node, the node never authenticates with the principal. This could be a problem if a program is diverted from it intended destination and invoked on a different node. The second problem is that there is nothing guarding against replay attacks. This is exacerbated by the first problem. Finally, we are using public key operations, which are notoriously slow.

To address these problems, we make use of a protocol in which a user and a node authenticate each other and generate a shared secret for future communications, identified by an SPI. The protocol is essentially a signed version of Diffie-Hellman with a few variations, and works as follows. The principal interested in authenticating itself sends its public key to the desired node asking to start the protocol. If the node wishes to authenticate with the requesting principal, it generates its public and private D-H values, signs them, and sends the response. On receipt of this message, the initiating principal generates its own values, signs them, and sends the message to the node. At this point, both parties can construct the shared secret. All messages in the protocol contain expiration timers and nonces to prevent replay, as well as exchange identifiers. These exchange identifiers are used at the conclusion of the protocol to create SPI's for future use of the shared secret.

Once the protocol is complete, parties may use the shared secret to authenticate via HMAC-SHA1 [8]. To prevent replay, each principal associates a counter with the shared secret. This counter monotonically increases with each message, so that any message that is received with a lower counter value is rejected. To deal with out-of-order delivery, a sliding window scheme may be used rather than a single counter value. We reflect the use of HMAC-SHA1 in PLAN by altering the signature of `authEval` to take a chunk and a tuple consisting of the the SPI, the counter, and the HMAC signature over all of the previously mentioned items.

4.2 Authorization

Once a PLAN program (more specifically, a chunk) has been authenticated, it must be authorized before it can use certain services. In our security architecture, each service is associated with with some level of privilege. These levels form a poset such that if a principal has a privilege level l, it may invoke services associated with level l, as well as services of level m such that $m \leq l$.

This approach has the advantage of scalability. If we have x principals and y protected services, we may only have to make $x + y$ associations – each service and each principal is associated with one level of privilege. A more naive implementation might associate up to y services with each principal, for up to $x \times y$ associations. Of course, we could encode this naive approach with ours by associating each principal with multiples levels of privilege.

We implement this approach in PLAN as follows. When invoked from a PLAN program, each protected service is required to call an authorization service, providing its level of privilege l_s as an argument. This authorization service queries the authentication service to obtain the current program's principal p, and then looks up that principal's level of privilege l_p (how that "looking up" is done will be explained shortly). If $l_s \leq l_p$ then the service invocation is approved.

To keep track of the privilege allowed to principals, we make use of the Query Certificate Manager [9] (QCM). QCM provides a means of describing trusted, distributed databases in the form of sets. Here, we use QCM to describe N sets of principals, where each set is associated with a node in the poset. The security relationship among the sets as described by

the poset is implemented via set inclusion in QCM. For example, if principals $p_1,\ p_2,\ \ldots\ p_n$ have privilege level l, and l is a child of m, then we specify set l as:

$$l\ =\ \{\ p_1,\ p_2,\ \ldots\ p_n\ \}\ \textbf{union}\ m$$

This essentially states that all principals allowed to access services at level l are those with privilege of exactly level l and those with levels higher than l. Using this framework, an authorization check for p to call a service with level l reduces to set membership test for p in that set; this is a fairly straightforward operation in QCM.

If used only to specify sets of principals on per-node basis, QCM is probably overkill. However, it has other properties that make it valuable. For one, sets described in a distributed manner impose no additonal query complexity on the client (which in our case is the authorization service). For example, a node A may define a set which refers to a set resident at another node B:

$$l\ =\ \{\ p_1,\ p_2,\ \ldots\ p_n\ \}\ \textbf{union}\ B.m$$

If the authorization service on A makes a membership test on set l, QCM will automatically query B if necessary. QCM may also makes use of certificates, which are signed assertions about set relationships, to short-circuit remote queries. This allows QCM to implement both *push*- and *pull*-based information-retrieval. One problem with using certificates in our current implementation is that they would need to be explicitly passed as arguments to an authorized service. This problem could be addressed by centralizing the authorization procedure, as described below. Finally, QCM should be useful in an active node for activities other than authorization; [3] describes a few examples. While we currently don't take full advantage QCM's features, we expect that QCM's distributed nature and simple interface will make it scale nicely as our needs increase.

Namespace-based enforcement

Requiring that each service perform its own authorization by calling the authorization service with its privilege level is simple, yet redundant in that each service is forced manage its own security. It would be convenient to instead centralize the mechanisms needed to do service authorization, but here the worry is that we might impose overhead onto unauthenticated programs. One approach is to make use of *namespace-based enforcement*. The idea here is that a program that wishes to access privileged services must call the authorization service itself. This service obtains the level of privilege allowed the program and then expands the program's namespace to include all of the services it is allowed to call. Once the program completes, the namespace is thinned to its original form.

A convenient way to implement this is to additionally perform authorization after authentication in the `authEval` service. To make use of QCM certificates, we could add them as a third argument to `authEval`. Once the chunk argument given to `authEval` finishes evaluation, the namespace can be thinned. There are a few sticky issues, but this approach seems promising, so we plan to explor it further.

5 Conclusions and Future Work

Active networks provide the opportunity to increase the quality, efficiency, and usability of the network. However, for this opportunity to be realized AN's increased flexibility must be tempered by improved safety and security. PLANet is an Active Network built with PLAN distributed programming system which makes use of *active packets* written in PLAN, and node-resident *services* written in OCaml. This paper has described a number of

changes to PLANet which improve its security. In particular, we have described a restriction to PLAN to reduce overconsumption of resources, and we have implemented CPU and memory counters to bound per-packet resource usage. We have also added to PLANet an infrastructure to protect access to services. This infrastructure uses a signed version of Diffie-Hellman to perform node-user authentication, where the generated shared secret is used with HMAC-SHA1 to sign future PLAN programs requiring authorization. Each principal is a member of a set where the sets are partially ordered based on privilege. We use the Query Certificate Manager to resolve queries about a principal's privilege.

We feel that these mechanisms are useful, but further design and implementation improvements are possible. In particular, the overhead of CPU and memory counters seems excessive. Furthermore, we would also like to further assess the scalability and usefulness of QCM as an authorization vehicle. Overall, we feel this work is satisfying as a preliminary study of security mechanisms in PLAN, and that it should lead to more interesting and well-developed ideas in the near future.

References

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

[2] M. Greenwald and D. R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, Seattle, Washington, 1998.

[3] Carl A. Gunter, Trevor Jim, and Bow-Yaw Wang. Authenticated data distribution using query certificate managers. `http:// www.cis.upenn.edu/ ~tjim/ papers/ qcm-abstract.html`, 1997.

[4] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. Planet: an active networking testbed. `http:// www.cis.upenn.edu/ ~switchware/ papers/ planet.ps`, 1998.

[5] Michael Hicks, Jonathan T. Moore, Pankaj Kakkar, Carl A. Gunter, and Scott Nettles. Network programming with plan. In *IEEE Internet Programming Languages Workshop*, Chicago, Illinois, 1998.

[6] Michael Hicks, Jonathan T. Moore, Pankaj Kakkar, Carl A. Gunter, and Scott Nettles. Plan: A programming language for active networks. `http:// www.cis.upenn.edu/ ~switchware/ papers/ plan.ps`, 1998.

[7] Eoin Hyden. *Operating System Support for Quality of Service*. PhD thesis, February 1994. Available as Technical Report No. 340.

[8] H. Krawczyk, M. Bellare, and R. Canetti. Hmac: Keyed-hashing for message authentication. Technical report, IETF RFC 2104, February 1997.

[9] Query certificate manager home page. `http:// www.cis.upenn.edu/ qcm`.