Chunks in PLAN: Language Support for Programs as Packets

Jonathan T. Moore Michael Hicks Scott Nettles
Department of Computer and Information Science
University of Pennsylvania
{jonm,mwh,nettles}@dsl.cis.upenn.edu

March 1, 1999

Abstract

Chunks are a programming construct in PLAN, the Packet Language for Active Networks, comprised of a code segment and a suspended function call. In PLAN, chunks provide support for encapsulation and other packet programming techniques. This paper begins by explaining the semantics and implementation of chunks. We proceed, using several PLAN source code examples, to demonstrate the usefulness of chunks for micro-protocols, asynchronous adaptation, and as units of authentication granularity.

1 Introduction

The current IP-based Internet has been a success in part due to the simplicity of its architecture. However, this success, while bringing greatly increased use, has also greatly increased the demand for new and more complex network services and protocols. The goal of Active Networking is to allow these demands to be met by increasing the flexibility with which the network can be changed. Active networking achieves its increased flexibility by making the network programmable.

In the SwitchWare project [2], we have been exploring how to make the network programmable both by allowing switches to be dynamically extended with new services and by allowing packets themselves to be programs. A key way in which we have explored the idea of packets as programs is through the design and implementation of PLAN, the Packet Language for Active Networks, a domain specific language for writing packet programs [8]. Using PLAN we have implemented PLANet [9], an internetwork in which every packet is a PLAN program, and in which all activities in the network proceed by evaluating PLAN packets. This paper draws heavily from examples developed while building PLANet.

Just because packets are programs does not mean that many of the familiar features of conventional packets do not need to be supported. In particular, the desire to build networks using layering requires that PLAN programs support encapsulation, while the need to support services such as checksumming and fragmentation means that PLAN programs must sometimes be treated as data. In this paper, we explore the key mechanism used in PLAN to support functionality like encapsulation and treating packets as data, a language construct called *chunks*.

Chunks consist of some code, a named function used as an entry point, and arguments for that function. When chunks are evaluated, they invoke the named function using the arguments. By transporting a chunk from one node to another before evaluating it, we achieve a kind of remote evaluation; such remote evaluation how PLAN supports the distributed communication and programming from which the network is built. Chunks are first-class data values, which means they can be used as function arguments (notably, as an argument in another chunk), returned from functions, stored in data-structures, and manipulated as bags of bits.

If chucks merely provided an exotic way of doing what can already be done they would be of limited interest. However, chunks, and the programs embodied in them, provide flexibility that goes beyond traditional packet headers. Besides showing how chunks support traditional networking functionality, we also show how chunks can be used in three additional broad ways: to support the implementation of *micro-protocols* [11], to provide *adaptive protocols* [6, 14], and to solve certain security problems in a novel way.

To provide context, we begin by presenting a brief background of PLAN and chunks in Section 2. Next, we discuss and provide examples of the applicability of chunks to micro-protocols, adaptive protocols, and security in Sections 3–5. Finally, we discuss related work in Section 6 and conclude with future research directions in Section 7.

2 Background

Our examples use PLAN and so here we provide a brief look at its features, focusing on chunks. The reader is referred to [8] for a more in-depth treatment.

PLAN is a simple, lexically-scoped language designed to be carried in packets and to fill and extend the role of traditional packet headers. To this end, PLAN's expressiveness is limited to permit active nodes to evaluate a PLAN program without requiring authentication: PLAN programs are functional in style and are guaranteed to terminate. Its syntax resembles that of ML.

To mitigate their limited expressive power, PLAN programs may invoke node-resident *services* that provide access to information (e.g., time of day, number of network interfaces, etc.) or to more powerful functionality (e.g., fragmentation, reliability, soft state, etc.). These services may be dynamically loaded over the network and may be written in a more standard high-level language like C, Java, or ML.

Every PLAN program contains a top-level "code hunk," or *chunk*. As mentioned, chunks have three components: PLAN code, an entry point function within that code, and bindings for the function's parameters. A chunk can be viewed as a snapshot of a traditional function call where the arguments have been assembled and the program is about to branch to the function itself. When a PLAN packet arrives at its destination node, its entry point function is called with the given arguments.

The most powerful aspect of chunks, however, is that they are first-class values in PLAN. The PLAN syntax

$$|f|(expr_1, ..., expr_n)$$

is an expression of type `chunk` which creates a new chunk. Intuitively, the $|$'s indicate the part of the expression whose evaluation is to be delayed: the function call itself. Specifically, the created chunk consists of the same code as the current execution, an entry point f, and bindings which are the values obtained by evaluating $expr_1, ..., expr_n$. Thus, chunks can be manipulated, copied, or passed as arguments, and, most notably, can even appear as bindings in other chunks. Indeed, encapsulating one chunk within another is a key mechanism we will use in the rest of the paper.

Besides treating it as data, the main action that can be performed on a chunk is to execute it: the `eval` service evaluates its chunk argument by loading the code segment and invoking the entry point function on its arguments. If we take the view, as above, that a chunk is a suspended function call, then `eval` resumes the computation, but with of course a different continuation.

To support such functionality as fragmenting or checksuming a chunk, we also provide the ability to create a concrete representation of a chunk in the form of a byte string. This representation is essentially the marshaled wire-format version of the chunk. A service related to `eval`, `evalBlob`, serves to evaluate this representation. Unlike the uses of code as data in Lisp-like languages, we so far we have found that uninterpretable strings of bytes suffice for our concrete representation.

```
svc verifyChecksum : (blob,int) -> bool
svc evalBlob : blob -> 'a

fun unchecksum(c:blob, sum:int): unit =
  if verifyChecksum(c,sum) then
    (evalBlob(c);())
  else
    () (* drop packet *)
```

Figure 1: Code for a checksum chunk

```
svc reassemble : (blob,int,bool,key) -> 'a

fun defrag(frag:blob, seqnum:int,
           morefrags:bool, session:key)
          : unit =
   (reassemble(frag, seqnum, morefrags,
               session); ())
```

Figure 2: Code for a fragmentation chunk

3 Micro-protocols

Most commonly-used protocols like TCP and IP are complex, with a variety of functionality and many options. Developing and testing such protocols can be difficult and error prone, and the resulting protocols are not particularly flexible. These problems have motivated past research on micro-protocols [13, 14]. Each micro-protocol embodies a single function or option; more complex behavior is achieved by composing these micro-protocols.

We will present here two micro-protocols: one for a packet checksum, and one for fragmentation. These protocols will also serve to show how chunks provide some basic networking implementation techniques within the context of packets as programs.

In PLAN, micro-protocols are composed through chunk encapsulation. In general, each micro-protocol takes a chunk plus additional arguments and returns one or more new chunks which add the micro-protocol's functionality. This is similar to encapsulation in traditional networking, where, as a packet moves down the network stack, each protocol layer encapsulates the higher-level packet while perhaps adding additional header information for itself.

For example, suppose we had a chunk c to which we would like to add checksumming. We can invoke a `checksum` service on c which converts it into a stream of bits (i.e., a "blob" type in PLAN) via the standard PLAN marshaling system, computes a checksum *sum*, and then wraps them in a chunk with a code segment like that shown in Figure 1. When this new chunk d is evaluated, `unchecksum` is called using c and *sum* as arguments. `Unchecksum` then calls the `verifyChecksum` service to ensure that c still has the proper checksum, and then either evaluates c or aborts, as appropriate.

As another example, consider the task of fragmentation; namely, we have some chunk c to transmit and evaluate remotely, but it may be larger than the MTU of the intervening path. We can have a `fragment` service which takes c (and the MTU size), represents it as a "blob," and divides it into MTU-sized[1] pieces. Each fragment is then wrapped in a chunk with a code segment as shown in Figure 2. The bindings for these new chunks would each have a piece of the original chunk, a sequence number, an indication whether it was the last fragment or not, and a unique identifier. The new chunks, when evaluated, simply register themselves with the `reassemble` service on the destination. This service collects all the incoming fragments, puts them in the proper order, reconstitutes the original chunk, and then evaluates it.

Figure 3 shows the composition of our two protocols to form a UDP-like delivery service. At the highest level, we have some data (represented as a `blob`) which we want to deliver to a specific port on a specific host. First we create the chunk c, which encapsulates this behavior. Then we create a new chunk d which adds checksumming. After querying the appropriate path MTU, we then invoke the `fragment` service to get a list of fragmentation chunks. If the original chunk was small enough to fit within one link-layer frame, these chunks take the form shown in Figure 4[2]. Finally, we use `foldl` to apply `send_frag`

[1] Less the overhead for the reassembly chunk.

[2] If fragmentation was actually required, we would have only a part of the innermost two chunks; however, for ease of illustration, we have not shown this case.

<table>
<tr><td>code</td><td><pre>fun defrag(frag:blob, seqnum:int,
           morefrags:bool, session:key) : unit =
  (reassemble(frag,seqnum,morefrags,session);
   ())</pre></td></tr>
<tr><td>entry point</td><td><code>defrag</code></td></tr>
<tr><td>bindings</td><td><code>frag =</code>
<table>
<tr><td>code</td><td><pre>fun unchecksum(c:blob, sum:int) : unit =
  if verifyChecksum(c,sum) then
    (evalBlob(c);())
  else
    () (* drop packet *)</pre></td></tr>
<tr><td>entry point</td><td><code>unchecksum</code></td></tr>
<tr><td>bindings</td><td><code>c =</code>
<table>
<tr><td>code</td><td>(empty)</td></tr>
<tr><td>entry point</td><td><code>deliver</code></td></tr>
<tr><td>bindings</td><td><code>p =</code> <port>
<code>b =</code> <data></td></tr>
</table>
<code>sum =</code> n</td></tr>
</table>
<code>seqnum =</code> 1
<code>morefrags =</code> false
<code>session =</code> <key></td></tr>
</table>

Figure 4: Chunk encapsulation

to send each fragment to *dest*.

When the fragments arrive they will be evaluated causing them to be placed in the reassembly table. Once all the fragments have arrived, they will be reassembled into chunk d, which when evaluated will verify the checksum, resulting in chunk c. When c is evaluated it will call `deliver` with port argument p causing data argument d to be delivered to the correct port.

Both of the above micro-protocols share the same basic structure: a service on the source is invoked with a chunk plus some configuration parameters. This results in the creation of a new chunk which carries the code to perform the destination side of the micro-protocol; note that this code may refer to services that reside on the destination (but need not necessarily reside on the source). The new chunk could then potentially be wrapped in yet another micro-protocol or simply sent across the network. At the destination, the chunks simply “unwrap” themselves.

This common structure makes many things easy. For one, we can have policy drive the composition of micro-protocols, rather than having dependencies built into complex protocols. Indeed, in our above example, rather than have fragments of a checksummed delivery packet, we could have invoked *fragment* first, and then done a `checksum` on each resulting chunk, thus ending up with checksummed fragments of the original chunk. Each micro-protocol takes a chunk and returns a chunk or list of chunks, so they may be

```
svc defaultRoute : host -> host * dev

fun send_frag (x:int*host,c:chunk)
              : int * host =
  (OnRemote(c,snd x,fst x,defaultRoute);
   x)

svc checksum : chunk -> chunk
svc fragment : (chunk,int) -> chunk list
svc getMTU : dev -> int
svc length : 'a list -> int
svc getRB : void -> int

fun udp_deliver (b:blob, p:port,
                 dest:host) : unit =
  let val c:chunk = |deliver|(p,b)
      val d:chunk = checksum(c)
      val p:host*dev = defaultRoute(dest)
      val ds:chunk list =
             fragment(d,getMTU(snd p))
      val l:int = length(ds) in
   (foldl(send_frag,
          (getRB()/l,dest),
          ds); ())
  end
```

Figure 3: UDP-style delivery

arbitrarily ordered in a type-correct way. Of course, the order *does* matter from a semantic point of view.

Secondly, micro-protocols may be coded to remove redundant functionality. For example, if a path only has Ethernet interfaces, the `checksum` service might simply return the original packet, as the checksum would be redundant with the underlying CRC check. Similarly, the `fragment` service can (and does) just return the original chunk if it was already small enough. Either of these optimizations remove the need to execute certain receiving code at the destination.

In fact, the destination will not even have to do a test to determine that it need not execute the receiving code! Since the demultiplexing path is encoded in the the way the chunks are encapsulated, the unnecessary code will simply not be called as an arriving chunk "unwraps" itself. This mechanism is in fact quite powerful and allows us to do straightforward asynchronous protocol adaptation, as we see in the next section.

4 Asynchronous Adaptation

Adaptive protocols are ones that can be dynamically reconfigured. In particular, they can react to changing network conditions to improve performance. For example, if a data stream is bottlenecked due to a low bandwidth link, it might be desirable to compress the stream. Similarly, many checksum errors arising from a noisy link might suggest using some sort of error correction scheme to introduce redundancy.

In most approaches to adaptive protocols, a primary problem is synchronization. Namely, a source and a destination must agree on the structure of the protocol stack they are using: a protocol where the sender encrypts data but the receiver fails to decrypt it would hardly be useful. As described in [14], such signaling protocols can often be complex (and sometimes expensive).

With PLAN chunks, there is no need for negotiation between the endpoints for correct functionality. A sender need only start using a new sequence of encapsulated chunks, and they will be correctly handled at the receiver because the structure of the "protocol stack" is *encoded in the packets themselves.* There need be no delay for the protocol switch to happen safely. Naturally, though, it may be important for a sender and receiver to communicate and cooperate to maintain an accurate network view so that a policy regarding the insertion and removal of micro-protocols may be reasonably applied.

Furthermore, adaptation with PLAN chunks is not limited to endpoints. It is straightforward to add micro-protocols just over some portion of the network infrastructure, as in the style of Protocol Boosters [6]. In this case, a router might intercept incoming PLAN packets, wrap their top-level chunks in a new "boosting" micro-protocol, and send them on to a "de-boosting" location. Once there, the wrapper chunk will perform the receive-side of the micro-protocol and then send the original top-level chunk

```
svc decode : blob -> (chunk * host)
svc getRB : void -> int
svc defaultRoute : host -> host * dev

fun decrypt(cd:blob) : unit =
  let (c,d):chunk * host = decode(cd)
  in
    OnRemote(c,d,getRB(),defaultRoute)
  end
```

Figure 5: Wrapper chunk for a virtual private network

on to its final destination. Conventional, non-active packets can be treated the same way, essentially letting them tunnel in an active packet to allow dynamic protocol composition.

We make these ideas more concrete by presenting two examples. Both involve micro-protocols which are applied at points within the network rather than just at the endpoints of a communication.

Virtual private networks. Our first scenario is that of virtual private networks, where we have several networks of trusted nodes that we wish to connect by traversing untrusted links. We would like to give all the end hosts the illusion of being within a single trusted network. This can be accomplished in a straightforward manner by encrypting and encapsulating packets between trusted networks. In the Internet, IPSec [4] may be used in exactly this way.

We can achieve a similarly elegant implementation using PLAN chunks. An end host transmits an unencrypted packet which is intercepted by a firewall machine when it is about to leave its trusted network. The firewall extracts the top level chunk and final destination from the packet, and then encrypts them using a secret shared with a corresponding firewall on the other side of the untrusted network. The firewall then creates a new wrapper chunk like that shown in Figure 5: the binding `cd` is the bit string resulting from the encryption.

When the wrapper arrives at the opposite firewall, it calls the `decode` service which accesses the firewall's shared secret to recover the original (unencrypted) chunk c and evaluation destination d. Finally, it sends c along to d. This scheme could be easily extended to encrypt additional fields (e.g., a nonce to prevent replay attacks).

Mobile computing. Our second scenario considers mobile computing over wireless links, which are more noisy than wire-based LANs. Wireless links often have poor TCP throughput, as negative acknowledgments due to checksum failures are interpreted (incorrectly) as network congestion. To compensate for packet errors, the networking software on the laptop could engage a forward error correction (FEC) micro-protocol when operating in mobile mode.

With PLAN chunks, we can easily limit the FEC just to the wireless link, thus conserving overall bandwidth in the rest of the (less lossy) network. On the source, we wrap our original chunk and its intended destination in a wrapper chunk which registers itself with the FEC service on the other side of the link. We would periodically generate an additional parity packet which also registers itself with the FEC service. In turn, the FEC service would verify the encapsulated original chunks and send them out to their final destinations.

One issue is that the laptop might cross a cell boundary, thus switching gateways. Normally, this would require some amount of synchronization and communication, but since the FEC chunks are carried with the packets, the new gateway immediately knows that forward error correction is being used by the laptop. At worst, the laptop may have to retransmit the batch of packets which were being transmitted when the switch occurred. Here, the fact that PLAN packets actively specify their processing saves us additional communication over the lossy link.

5 Security

PLAN was designed as a restricted language with the intention that programs using only services that provide appropriate protection would not require authentication. In fact, other active networking research [1] has shown that authenticating every active

```
svc authEval : (chunk,blob,blob) -> 'a

fun wrap (c:chunk, sig:blob, id:blob)
         : unit =
  (authEval(c,sig,id); ())
```

Figure 6: Wrapper chunk for authorization

packet results in unacceptable performance degradation.

On the other hand, for full advantage to be taken of the flexibility of active networking, we want to allow PLAN programs to use services that require that packets be authorized. Functions about which very few *a priori* claims can be made suggest the need for some sort of authorization policy controlling their installation and use.

PLAN chunks provide an elegant mechanism for this sort of authorization. A user with greater-than-normal privilege could take a chunk and compute a cryptographic signature for it. In turn, the chunk and signature could be encapsulated in another chunk like the one shown in Figure 6. When evaluated, the `authEval` service verifies that the signature matches the chunk and user's identity (represented by a public key), thus *authenticating* the user. The service then evaluates the chunk in an environment appropriate to the user's level of authorization. The advantage of this approach is that in signing a chunk, the user can define exactly the scope in which greater privilege is needed: the duration of the evaluating the chunk. An alternative certificate-based approach is potentially less precise.

Of course, this scheme only permits a user to authenticate himself to an active node, but not vice versa. There is also a vulnerability to replay attacks, and public key cryptosystems are notoriously slow. Fortunately, all three of these problems can be addressed, as described by the Secure Active Network Environment (SANE) [3], and implemented in our current testbed, PLANet [7].

6 Related Work

Micro-protocols are used in the x-kernel [11, 13]. However, the stress there is on software engineering and code reuse; although their approach enables the relatively simple development of new protocols, they must still be composed statically, whereas micro-protocols implemented with PLAN chunks can be dynamically reordered.

Ensemble [14] is a toolkit for distributed application development. Unlike the x-kernel, applications may adapt and dynamically reconfigure their protocol stacks. However, Ensemble uses a Protocol Switch Protocol which halts communication, synchronizes through a central coordinating participant, and then resumes communication. PLAN chunks do not require this pause for synchronization and do not need centralized coordination. Furthermore, PLAN chunks are not limited to an end-host only regime of operation.

Protocol Boosters [6, 12] interpose additional functionality within the network infrastructure. These boosters enhance performance in a way that is transparent to the applications communicating across the "boosted" subnets. However, for multi-component boosters, signaling is required to support the addition or removal of a booster. Finally, because they reside in the network infrastructure itself, some boosters are subject to failures due to routing changes sending boosted packets around their intended de-boosting element. PLAN chunks are not subject to these failures because the chunk encapsulation essentially records which micro-protocols have been applied and must be undone at the destination.

Application-specific safe message handlers (ASHs) [15] also support dynamic protocol layering in the Exokernel [5]. Dynamic code generation is used to allow integrated layer processing based on upcalls. Since PLAN chunks are carried in packets, it is not clear that dynamic code generation or just-in-time compilation will provide an advantage over simple interpretation. Furthermore, the evaluation of PLAN chunks already naturally suggests an upcall-based organization. Lastly, PLAN chunks are not limited to end hosts.

Other active networking projects such as

Netscript [17] and ANTS [16] offer flexible networking as well. In particular, Netscript permits dynamic recomposition of protocols, but still requires synchronization across nodes to make sure that protocol stacks match correctly. It is not clear whether ANTS protocols are composable as their programming abstractions ensure that protocols cannot interfere with one another.

7 Future Work and Conclusions

One of the drawbacks of our current implementation is that PLAN evaluation is currently quite costly. Partially this is a result of lack of tuning, and the use of a byte-code interpreted language implementation. A faster PLAN interpreter and service routines might be constructed using certified run-time code generation techniques as in the Cyclone [10] compiler.

A general concern about our approach is the space cost of carrying the code in the packet. To mitigate this overhead, we are currently considering ways in which the participants in a protocol may cache code rather than always transmitting it with the packet. One promising approach is to add language-level *remote-references* that may be thought of as pointers to remote objects. Since all PLAN values (including chunks) are *immutable*, the contents of a remote reference may be safely cached without the need for a coherence protocol. This approach might also allow us to cache some of a chunks bindings as well, which might result in further space savings.

Implementation of various Protocol Boosters should be straightforward using PLAN chunks. Given a more streamlined PLAN infrastructure, it should be possible to experimentally demonstrate the utility of these techniques as well as other adaptive protocols. In particular, it would be intriguing to boost regular TCP/IP traffic across a PLAN-based subnet.

In this paper, we have described chunks, a construct of our domain-specific programming language, PLAN, and have provided several examples of their use. Firstly, chunks representing micro-protocols can be arbitrarily ordered and composed to create more complex protocols. Secondly, since the encapsulated chunks are carried in the packets themselves, they permit asynchronous protocol adaptation. Finally, chunks provide a convenient level of granularity for program authentication. Indeed, chunks are a powerful enough abstraction to permit the elegant expression of a number of powerful networking mechanisms.

References

[1] D. S. Alexander, Kostas G. Anagnostakis, W. A. Arbaugh, A. D. Keromytis, and J. M. Smith. The Price of Safety in an Active Network. Technical Report MS-CIS-99-02, University of Pennsylvania, January 1999.

[2] 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. The SwitchWare Active Network Architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3), May/June 1998.

[3] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromytis, and Jonathan M. Smith. A Secure Active Network Architecture: Realization in SwitchWare. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, May/June 1998.

[4] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, August 1995.

[5] Dawson R. Engler, M. Frans Kaashoek, and Jr. James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[6] D. C. Feldmeier, A. J. McAuley, J. M. Smith, D. Bakin, W. S. Marcus, and T. Raleigh. Protocol Boosters. *IEEE JSAC, Special Issue on Protocol Architectures for the 21st Century*, 16(3):437–444, April 1998.

[7] Michael Hicks. PLAN System Security. Technical Report MS-CIS-98-25, University of Pennsylvania, July 1998.

[8] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 86–93, September 1999.

[9] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl Gunter, and Scott Nettles. PLANet: An Active Internetwork. In *IEEE Conference on Computer Communications (INFOCOM)*, March 1999. To appear.

[10] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. In *ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, January 1999.

[11] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[12] A. Mallet, J. D. Chung, and J. M. Smith. Operating Systems Support for Protocol Boosters. In *HIPPARCH Workshop*, June 1997.

[13] Sean W. O'Malley and Larry L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[14] Robbert van Renesse, Ken Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building Adaptive Systems Using Ensemble. Technical Report TR97-1638, Cornell University, July 1997.

[15] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. In *ACM Communication Architectures, Protocols, and Applications (SIGCOMM'96)*, August 1996.

[16] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying net work protocols. In *IEEE OPENARCH*, April 1998.

[17] Y. Yemini and S. da Silva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, L'Aquila, Italy, 1996.