INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Safe and Efficient Active Network Programming

Scott Thibault, Charles Consel, and Gilles Muller

N ˚ 3355

Février 1998

THÈME 2

Rapport de recherche

ISSN 0249-6399

Safe and Efficient Active Network Programming

Scott Thibault, Charles Consel, and Gilles Muller*

Thème 2 — Génie logiciel
et calcul symbolique

Projet COMPOSE

Rapport de recherche n 3355 — Février 1998 — 17 pages

Abstract: Active networks are aimed at incorporating programmability into the network to achieve extensibility. An approach to obtaining extensibility is based on downloading router programs into network nodes. Although promising, this approach raises several critical issues: expressiveness to enable programmability at all levels of networking, safety and security to protect shared resources, and efficiency to maximize usage of bandwidth.

This paper makes contributions to all three issues above.

Expressiveness. We have extended PLAN, an active network language for diagnostics, so that the language can express *application protocols*. These extensions are illustrated with two protocols: adaptive compression and active bridging.

Safety and security. To address these critical aspects, we give examples of properties of PLAN programs that can be *automatically* checked due to the use of a restricted language.

Efficiency. We show that an automatically generated *run-time* compiler for PLAN and our extensions produces code with similar performance to an equivalent compiled Java program. Measurements are presented for an active bridge.

Key-words: Active netorks, Domain-specific languages, partial evaluation, run-time code generation, Ethernet bridge.

(Résumé : tsvp)

* {sthibaul}{consel}{muller}@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Conception de programmes sûrs et efficaces dans les réseaux actifs

Résumé : Les réseaux actifs sont des réseaux caractérisés par la propriété d'être extensibles. Une solution au besoin d'extensibilité repose sur le chargement de programmes de routage dans les nœuds du réseaux. Bien que prometteuse, cette approche soulève de nouveaux problèmes de recherches : (i) puissance d'expression pour permettre de traiter des protocoles de différents niveaux, (ii) sûreté et sécurité pour protéger les différentes ressources partagées (réseaux, nœud,...), et (iii) efficacité afin de maximiser l'utilisation de la bande passante du médium.

Dans cet article, nous décrivons des solutions aux trois questions précédentes :

Puissance d'expression. Nous avons étendu PLAN, un langage dédié au diagnostic dans les réseaux actifs, de telle manière qu'il soit possible de décrire des *protocoles spécifiques à une application*. Les extensions proposées sont illustrées au moyen de deux exemples : un protocole de routage intégrant un mécanisme de compression adaptatif et un répéteur Ethernet intelligent.

Sûreté et sécurité. Pour adresser ces aspects critiques, nous montrons que grâce à l'utilisation d'un langage restreint il est possible de vérifier *automatiquement* un ensemble de propriétés.

Efficacité. Nous montrons qu'il est possible de générer automatiquement un compilateur à la volée à partir d'un interprèteur de notre langage. Les programmes objets produit par ce compilateur sont aussi performants que ceux produit par un compilateur Java optimisant. Pour évaluer précisément notre approche, nous comparons plusieurs implémentations d'un répéteur Ethernet intelligent.

Mots-clé : Réseaux actifs, langages dédiés, évaluation partielle, génération de code à l'exécution, répéteur Ethernet.

1 Introduction

Active networks [12] enable rapid network evolution by making routers programmable. Programs can be downloaded into network routers to control the processing of packets. A router program can be associated to a packet, an application or some other entity. As a result, the network becomes flexible and can adapt to specific needs. Also, it is extensible since new functionalities can be introduced dynamically.

The introduction of highly-programmable networks, however, raises a number of issues [12]. First, safety and security should be ensured because network routers are shared resources. Second, given the heterogeneity of the network, the module executing router programs should be highly portable. Domain-specific languages (DSL) are aimed at designing languages suitably restricted to enable strict constraints to be enforced and specific properties to be determined. In fact, most existing approaches to active networks are based on DSLs. An example of such a DSL, of particular interest for the present work, is PLAN [5]. This language is concerned with programming network routers for diagnostics. Safety and security are achieved by restricting the semantics of the language: it is a first-order, strict functional language — essentially a subset of ML [5]. To achieve flexibility, the primary motivation of active networks, PLAN is defined in two levels: first, a PLAN program represents the glue between coarse-grained operations, called *services*; second, services are node-resident primitives which are intended to be written in a general-purpose language.

Some level of safety and security are achieved because of specific features of the language. Mainly, strong typing ensures type safety and non-interference of concurrent processes; absence of recursion (and loops) guarantees termination (for terminating services). Furthermore, portability is obtained by an implementation in Java for both services and the PLAN interpreter.

PLAN has demonstrated that a DSL-based approach for active networks is a promising direction. This work also suggests important challenges:

Applicability. Although PLAN has been successfully used for network diagnostic applications, its usability needs to be demonstrated on a wider range of networking purposes.

Safety and security. The restrictions of PLAN, combined with its two-level implementation (program and services), has proved to ensure important safety and security properties. Other safety and security issues still need to be addressed such as global termination and packet duplication.

Efficiency. As noted by Hicks *et al.* [5], the performance of PLAN processing needs to be improved. The authors mention the fundamental conflict between sending the PLAN source program to nodes to allow them to do defensive checking for safety and security reasons, and sending lower-level representations making defensive checking difficult, if not impossible.

This paper makes contributions to all the challenges listed above.

PLAN-P: an extension for application protocols. The first contribution of this paper aims at broadening the applicability of PLAN. Indeed, this language, as described in [5], mainly targets diagnostic network applications. We have extended the language to make it possible to express protocols. More precisely, we introduce the concepts of packets as first-class objects, persistent state, and channels in response to specific needs of protocol development.

Strengthening safety and security in PLAN. As advocated by promoters of DSLs, because of their simple semantics and their restrictions, such languages make it possible to determine properties typically undecidable in general-purpose languages (*e.g.,* termination). We propose to strengthen the safety and security of PLAN(-P) to guarantee that: packets do not cycle within the network (without a run-time resource bound), all packets are delivered, and packet duplication is linear. Each of these properties can be automatically checked due to certain restrictions or domain-specific attributes of the language which would otherwise make automatic proof impossible.

High-performance PLAN execution via run-time specialization. The third contribution of our work is in improving the performance of PLAN programs, and more generally programs which require late checking. As mentioned above, PLAN source programs need to be made available to network routers in order for them to perform defensive checking. Downloading the source requires that PLAN programs be interpreted, which is too slow, or compiled, which is too costly. Of course, one can imagine developing forms of *just-in-time* (JIT) compilers for PLAN. However, regardless of the compilation technology, the main problem with this approach has to do with the heterogeneous nature of the network which requires a compiler to be available for each architecture.

To solve the conflict between late program checking and performance we propose to use specialization. More specifically, and like Hicks *et al.*, we use an interpreter to process PLAN programs. Our interpreter is written in C to be portable on all routers. However, unlike the existing implementations of PLAN, our interpreter is specialized with respect to a PLAN program in order to eliminate the interpretive overhead. Because the program is bound late, a run-time form of program specialization is used. Our approach relies on a program specializer for C called Tempo [2]. Unlike other existing program specializers, Tempo is not limited to specializing programs at compile time; it is also capable of specializing programs at run-time. That is, it can specialize a program with respect to values which are not available until run-time.

As discussed in section 5, our experiments show that the PLAN-P interpreter specialized with respect to a learning bridge program [1] achieves up to 85% of the throughput obtained with an equivalent bridge written in C. More importantly, we implemented a bridge in Java to assess the performance of a highly-portable language. We compiled this bridge with an optimizing off-line Java compiler [7] to get the best possible performance. When compared to our specialized PLAN-P interpreter, the compiled Java bridge has similar performance. In comparison to results on the Pentium architecture presented by Alexander *et al.* [1], we achieve twice as much throughput.

These performance measurements demonstrate that program specialization is a key technology to enable simultaneously late checking of source programs and efficiency. Furthermore, efficiency is obtained without sacrificing portability since an interpreter is used to implement our extension of PLAN. Interpreters also permit prototyping and are easier to develop and maintain.

2 The PLAN-P Language

PLAN is a domain-specific language developed for active networks. The language was initially designed for diagnostic network applications with the following goals: it should be flexible, safe, secure, efficient, lightweight, and usable. The language is a first-order, strict, strongly-typed functional language with restricted primitives and data types.

Although expressions are essentially a subset of ML, the execution model of PLAN is one of distributed computation. A PLAN active network replaces traditional packets, IP header plus data, with packets whose header is a PLAN program and whose data are inputs to the program. Distributed computing is supported with a remote evaluation construct. Remote evaluation is implemented by constructing a new packet with the source of the program to be remotely evaluated and the arguments of the procedure to be invoked on the remote host.

2.1 PLAN-P: An Extension of PLAN for Protocols

One goal of this work is to expand the domain of application of the PLAN language. In particular, we wish to develop application protocols. Our main extensions to PLAN consist of the following three notions which directly correspond to three important needs for writing application protocols.

- Packets as first-class objects - since protocols are concerned with the collective processing of a group of packets we must associate a program with multiple packets rather than a single packet, as in PLAN. From this view, a program is not a packet but rather a program that processes packets.

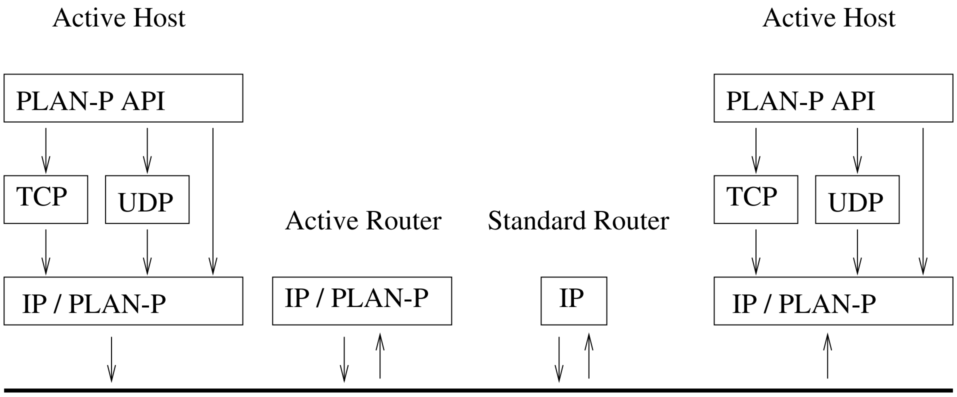

Figure 1: PLAN-P architecture

- Protocol state - many adaptive protocols, such as the learning bridge presented later, must maintain information about its current state in order to process subsequent packets. Thus, we must have a state that persists across packets.
- Channels - to manage complexity of protocols we need to be able to reason about both the local (node) and global (network) packet forwarding behavior. Channels associate a state to each packet which provides a state-transition view of the global behavior of a packet traveling through the network.

Figure 1 depicts the PLAN-P architecture. This architecture is based on IP and allows PLAN-P programs to process existing IP/UDP/TCP packets and create new packet types on top of these existing protocols. PLAN-P programs are injected into the network via the PLAN-P API and can be associated with 1) an existing protocol, 2) a new PLAN-P protocol based on an existing protocol, or 3) a socket. The following two sections describe example PLAN-P programs for cases 1 and 2. The third case is similar to case 2 with the addition of extra information in the packet to indicate which socket/machine the packet belongs to.

2.2 Adaptive Compression Example

Figure 2 shows an example PLAN-P protocol. This example utilizes multiple channels for a compression protocol which adapts to the nodes in the network. Each channel defines a corresponding function of three arguments: the current state of the protocol, the current state of the channel, and a packet. The function body defines the channel behavior by altering and forwarding the packet, and returning new values of the protocol and channel state.

The packet type determines which basic protocol the program applies to and, for new protocols, includes additional types for the protocol specific packet state. The compression example defines a new protocol built on IP as indicated by the declared packet type (`ip*blob`). The `blob` type is an opaque type which represents arbitrary sized binary data. A protocol based on TCP might have a packet type like `tcp*blob`, in which case the IP header is ignored and unchanged in forwarded packets. By declaring the packet type to be `ip*tcp*blob`, the program can access and modify both headers. Finally, in order to implement new protocols we can extend the header information. For example, defining the packet type as `ip*int*blob` includes an integer field in each packet in addition to the IP header. When multiple channels are defined, an additional state is implicitly associated to packets that indicates which channel the packet belongs to. Thus, multi-channel protocols, like the compression example, define a new protocol with an extended packet type.

The local execution of a protocol is as follows. The protocol is identified by a unique key stored in the IP header as an IP option as described by Wetherall and Tennenhouse [15]. The implicit state of the packet, identifying the channel it belongs to, is extracted from the packet and the corresponding function is invoked with the current state

```
channel uncompressed(ps : unit, cs : unit, p : ip*blob) : unit*unit =
  if thisHost() <> ipDst(fst(p)) then
    let
      val next : host = defaultRoute(ipDst(fst(p)))
    in
      if linkCapacity(next) < compressionThreshold then
        (OnNeighbor(compressed,next,(fst(p),compress(snd(p)))); (ps,cs))
      else
        (OnNeighbor(uncompressed,next,p); (ps,cs))
    end
  else
    (deliver(IP,p); (ps,cs))

channel compressed(ps : unit, cs : unit, p : ip*blob) : unit*unit =
  if thisHost() <> ipDst(fst(p)) then
    (OnNeighbor(compressed,defaultRoute(ipDst(fst(p))),p); (ps,cs))
  else
    (deliver(IP,(fst(p),uncompress(snd(p)))); (ps,cs))
```

Figure 2: Adaptive compression protocol

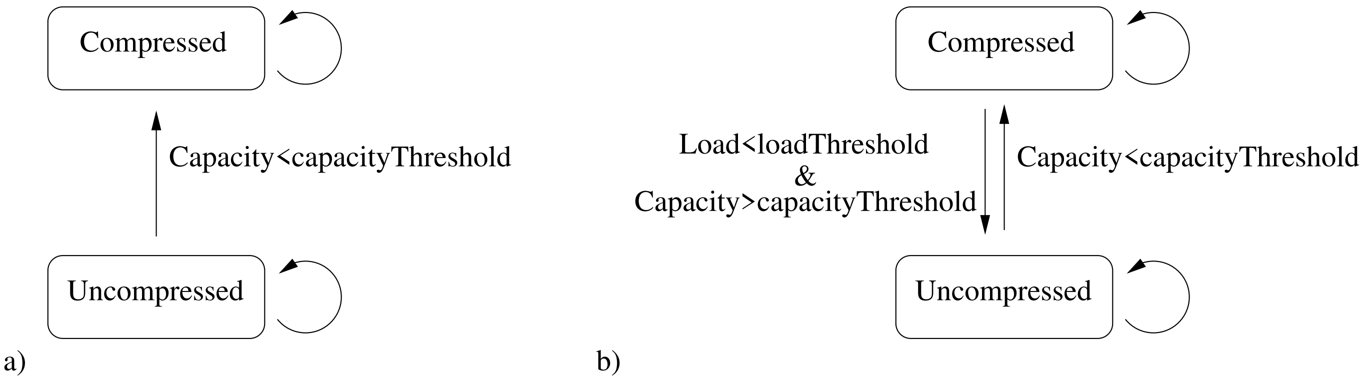

Figure 3: Channel view of a) one-way compression protocol and b) two-way compression protocol

(protocol and channel) and the packet as arguments. The returned value is used to update the current state and processing is complete. Packet forwarding is performed in the channel functions with the built-in primitives `OnRemote` and `OnNeighbor`.

The behavior of the compression example is simple. Packets are initially sent on the `uncompressed` channel. This channel's behavior is, at each intermediate node, to inspect the transmission capacity of the link to the next node, and if the capacity is below some threshold, then the packet is compressed and forwarded on the `compressed` channel[1]. The first argument of `OnNeighbor` indicates the channel on which the packet is forwarded. The functions `thisHost`, `defaultRoute`, `ipDst`, `linkCapacity`, `compress`, and `deliver` are language primitives. The availability of application specific primitives, such as `compress`, can be ensured by the initialization process.

The behavior of the `compressed` channel is to continue forwarding the packet until it reaches its destination. The packet must then be uncompressed before it is delivered. The global behavior of a packet can be considered by viewing

[1]One might also adapt to the processing load of the intermediate node.

```
val timeLimit : int = 1000000

channel system(ps : unit, cs : (interface*time) hash_table, p : ethernet*blob)
initstate mkTable(100) is
if isBroadcast(etherDst(fst(p))) or isMulticast(etherDst(fst(p))) then
  (simpleRepeat(p); (ps,cs))
else
  (insert(cs,hashKey(etherSrc(fst(p))),(thisInterface(),getTime()));
   try
     let
       val entry : interface*time = find(cs,hashKey(etherDst(fst(p))))
     in
       if (getTime() - (snd(entry))) < timeLimit then
         (OnNeighbor(system,fst(entry),p); (ps,cs))
       else
         (simpleRepeat(p); (ps,cs))
       end
   handle NoEntry => (simpleRepeat(p); (ps,cs))
  )
```

Figure 4: Extract of learning bridge

the channels as states in a state machine. For example, the behavior of the example is depicted in figure 3a. Although, not shown in the example, one might also want to adaptively uncompress packets at intermediate hops instead of only at the destination in order to reduce loads on the end node. In this case, the behavior might be as depicted in figure 3b.

2.3 Learning Bridge Example

A bridge is a network node which is connected between multiple LANs to form one logical LAN. A simple bridge repeats every packet received on all other LANs connected to the bridge. A learning bridge keeps track of where packets come from in order to determine which LAN a host is connected to, so that packets for that host are only repeated on the LAN it is on.

Figure 4 shows an extract of a PLAN-P implementation of the active bridge described by Alexander *et al.* [1]. This example uses the predefined channel `system` to modify the behavior of an existing protocol. Packet processing for `system` channels is as follows. The type of the packet is determined from the packet header, and if a `system` channel exists for this packet type, the corresponding function is invoked as for standard channels. Since the `system` channel is used to add behavior to existing protocols, the packet type must be a standard packet type (*i.e.*, `ip*blob`, `ip*udp*blob`, *etc.*).

In the learning bridge example, we define the channel state to be a hash table in order to store the interface (an identifier for the LAN) on which packets from each host are received, as well as the time the entry was created. Since this is a bridge between Ethernet LANs, the type of the packet argument is `ethernet*blob`, where `ethernet` is a built-in type for Ethernet headers and `blob` is an opaque type representing some binary data.

In the body of the function, if the packet is a unicast packet then we insert an entry in the hash table for the host the packet came from (`etherSrc()`) with the interface it was received on (`thisInterface()`) and the current time (`getTime()`). This is the learning portion. To forward the packet, we look for an entry in the table for the destination host of the packet. If an entry is found and it has not expired, the packet is forwarded on the interface in the table. The `simpleRepeat` function, not shown, repeats the packet on all interfaces except the one the packet was received from for the cases where no entry is found or the entry has expired. The functions `mkTable`, `etherDst`, `isBroadcast`, `isMulticast`, `hashKey`, `thisInterface`, `getTime`, `insert`, and `find` are language primitives.

3 Safety and Security

Adding programmability to a system, as in active networks, is a general solution to the problem of providing extensible and adaptable interfaces between systems. A disadvantage of this approach, however, is that it introduces additional security risks [12]. Security is particularly important in the context of active networks because the system is shared by many users and the users are typically not the owner of the network. In addition to security, we are also interested in the safety of active network programs. Due to the distributed nature of network programming and the difficulty of debugging, safety is also important for active networks. Furthermore, techniques to overcome security problems often merely transform the problem to be one of safety. For example, a resource bound, like the time-to-live field of IP or that of PLAN [5], solve the security problem of packets infinitely looping and consuming resources, but transforms it into one of safety: packets may be unintentionally terminated.

PLAN and PLAN-P are examples that demonstrate how DSLs can provide programmability while at the same time satisfying security and safety demands. The key idea is to restrict the language in such a way that the desired properties are satisfied *a priori*, or can be automatically checked by program analysis. One can view a DSL as providing a "glue" language which allows domain specific behaviors, in the form of built-in primitives and constructs, to be combined to form different behaviors. Since the primitives are semi-fixed, it is feasible to prove properties about them by hand once, and then properties about DSL programs can be more easily proved as composition of these properties. DSLs also provide the benefit of using established techniques in programming languages such as formal definitions of languages and analysis frameworks (*e.g.,* type systems). The remainder of this section presents three program properties that can be automatically checked due to the use of a DSL. They apply not only to PLAN-P but equally as well to PLAN.

Global termination. First we consider the termination of PLAN programs. PLAN was originally developed to have the property that execution of a PLAN program on a given node is guaranteed to terminate. This is a direct result of restricting the language to not allow recursion or unbounded loops. Recursive calls, however, are allowed for remote calls and thus, a packet may cycle forever on the network. One solution to this problem is to introduce a resource bound which is decremented on each hop as the time-to-live field of IP [5]. This is not entirely satisfying because it introduces a safety problem of unintended program termination. However, by making an assumption that the IP routing tables do not contain cycles, we can use the knowledge about the domain to prove that PLAN(-P) programs do not cycle. An outline of the proof strategy is presented in appendix A.

Guaranteed packet delivery. Another property that can be statically checked is that all packets are delivered. To prove this safety property, we assume that the underling network is reliable (*i.e.,* it does not lose packets)[2]. The basis of the proof is as follows. If packets are guaranteed not to cycle (as proven above), the program handles all exceptions (*i.e.,* it can not terminate due to an unhandled exception), and packets are forwarded for all execution paths (*i.e.,* the program does not intentionally drop packets), then we are guaranteed that the packet will be delivered.

Checking that all exceptions are handled can be easily achieved by restricting the `handle` clause to statically named exceptions. This restriction is useful since many primitives raise exceptions and can be easily forgotten. Finally, since the only way to forward a packet in PLAN-P is via `OnRemote` or `OnNeighbor`, it is easy to check that all packets are forwarded by checking for the existence of either of these statements on all possible execution paths.

[2]Since we are interested in the safety of the *program* and not the network, this assumption is appropriate.

Safe packet duplication. Finally, it is possible to verify that packets are not duplicated in an exponential manner. As a first attempt, we can simply verify that there is no execution path with more than one call to either `OnRemote` or `OnNeighbor` (*i.e.,* there is no duplication). However, this strategy is too conservative. We can extended it by verifying that for all execution paths there exists at most one `OnRemote` or `OnNeighbor` statement whose channel argument is one that might create copies of a packet. This property is proved using a standard fix-point iteration technique. This is less conservative and allows, for example, programs which send information back to its source as it travels.

4 Efficient PLAN-P Execution via Run-Time Specialization

As discussed earlier, programmable networks require late checking of programs. As a result, a representation close to the source program must be used. This constraint naturally suggests the use of an interpreter to run programs. In fact, this is the approach used for PLAN where the interpreter is written in Java to ease portability [5]. However, as noted by Hicks *et al.* [5], this double layer of interpretation calls for performance improvement.

Our approach to address this critical issue is to use program specialization. This program transformation is aimed at specializing an interpreter with respect to a given PLAN-P program. As a result, the interpretation layer can be removed and efficiency can be achieved.

4.1 Program Specialization

The idea of program specialization is to instantiate a program with respect to known inputs. This technique performs aggressive constant propagation and constant folding inter-procedurally. In the present case, it specializes an interpreter with respect to a given program. This transformation results in performing all the operations which depend on the program argument of the interpreter.

Traditionally, program specialization is performed at compile time. However, Consel and Noël extended this technique to allow programs to be specialized at run time [4]. In their approach, given a program and a description of the known inputs (not the actual values), templates which represent the building blocks of all possible specialized programs, are automatically generated. In addition, a run-time specializer is automatically generated which performs the computations relying only on known inputs, selects binary templates, and fills template holes with actual values. The templates and run-time specializer are compiled and linked *prior* to run-time. The template approach uses existing compilers for portability. Noël *et al.* demonstrate the low cost of run-time specialization and the efficiency of the run-time specialized code on various scientific and graphics programs [6]. In fact, the specialization costs for the PLAN-P interpreter only require the specialized program to be executed once or twice to be amortized (*i.e.*, it takes less time to specialize the interpreter and execute the result twice than it does to interpret the program twice).

4.2 Tempo

This approach to run-time specialization has been implemented and integrated in a program specializer for the C programming language. It performs specialization both at compile time and at run time [2, 3, 6] It has been successfully used for various applications ranging from systems programs to scientific code [2]. For example, Tempo has been applied to the XDR layer of the SUN Remote Procedure Call (RPC). The specialized version of the RPC is up to 3.5 times faster than the original XDR layer and up to 1.5 times faster than the original RPC for a round-trip [8, 6]. Tempo operates on both the SPARC and Pentium architectures under Solaris and Linux.

It is critical that Tempo allows programs to be specialized at run time in order to enable late program checking. Since the PLAN-P interpreter does not change, its run-time specializer can be generated once and for all. A specialized version is produced whenever a PLAN-P program is received by the router node. In fact, run-time program specialization preserves the advantages of an interpretation-based approach while allowing performance comparable to compiled code, without requiring the development of a compiler. In essence, run-time specialization gives us the functionalities of a *portable* JIT compiler for the price of an interpreter. These advantages are important since DSLs evolve frequently, and the network is heterogeneous.

4.3 The PLAN-P interpreter

The structure of the interpreter for PLAN-P follows the methodology defined by Thibault and Consel [13]. It consists of dividing the interpreter into two parts: the top part is concerned with operations dependent on the input program whereas the bottom part includes operations dependent on the program's input values. Assuming the program to be known at specialization time, the top part of the interpreter is eliminated (the interpretation layer) whereas the bottom part (the dynamic computations) remains. Thibault, Marlet, and Consel successfully applied their approach to the automatic generation of efficient video device drivers [14].

One of the advantages of dividing an interpreter into two parts is to clearly identify *a priori* the interpretation layer and the dynamic computations. This division is then checked by Tempo's dependency analysis which propagates the description of the known inputs and visually displays the results. As a result, the degree of specialization, and thus the optimization level of the interpreter is predictable.

Since Tempo can treat realistic C programs, our PLAN-P interpreter has a very natural structure; it is not necessary to write contorted programs to make them amenable to program specialization. It is defined as a recursive procedure traversing the program to be executed. More importantly, the PLAN-P interpreter specializes very successfully. The specialized interpreter typically runs 40 times faster than the interpreter clearly demonstrating the high degree of specialization of the PLAN-P interpreter.

5 An Active Bridge Experiment

In this section, we describe an experiment used to assess the performance of programs produced by the run-time specializer. We take as an example the active bridge. A bridge is a node which connects two or more LANs together to form an extended LAN. In our experiment, we implement the learning algorithm described by Alexander *et al.* [1]. The learning algorithm uses a hash table to record the source address and the LAN of received packets[3], thus learning which hosts are on which LANs rather than repeating packets on all LANs. We chose this application because it emphasizes the requirement for good performance and has already been accepted in the context of active networks.

5.1 Experimental Context

The configuration used for the experiment was two hosts connected to a bridge via 100 Mbps Ethernet. Both hosts and the bridge were 167 MHz Sun Ultra 1 Model 170s with 128Mb of main memory, 16kb of instruction cache, and 16kb of data cache. The operating system was Solaris 5.5.

For latency and throughput measurements, we give the performance of a simple repeater written in C to provide an upper bound for a user mode process. Since the PLAN-P system has not yet been integrated in the kernel, the bridge software was run at user level using the Solaris stream interface to access the Ethernet devices. In addition to the C repeater, results are presented for three programs which implement the learning bridge with the same algorithm implemented in C, Java, and PLAN-P. The C version represents the upper bound of the performance one could obtain with the particular algorithm used. The Java version was compiled using Harissa, an off-line optimizing Java compiler [7]. It represents the upper bound of the performance of the Java/JIT approach to mobile code. The PLAN-P version is our PLAN-P interpreter specialized at run-time with respect to the bridge program.

[3]The current time is also recorded in order to adapt to network changes by expiring old entries.

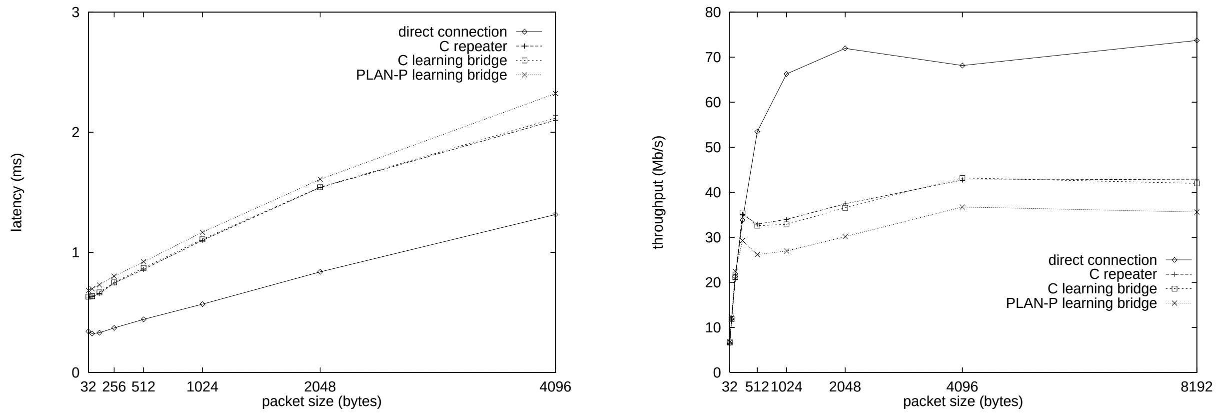

Figure 5: Latency (left) and Throughput (right)

5.2 Micro-benchmark

The purpose of the micro-benchmark was to establish the per packet processing overhead of the C, Java, and PLAN-P programs. The micro-benchmark consisted of pre-loading a packet trace into memory and measuring the execution time of processing each packet. The average per packet execution times are as follows:

Program	**Time (μs)**
C	4.7
Java	19.2
PLAN-P	18.8

As can be seen in the table, the performance of the Java and PLAN-P versions are essentially the same, whereas the hand coded version is roughly 5 times faster. One reason is that both the Java and PLAN-P versions use generic hash tables provided with the language and the hash table used in the C version was specifically written for the bridge. The Java version additionally has overheads due to object manipulation and run-time bounds checking. The PLAN-P version has additional overheads due to intermediate copies that are not optimized at run-time and lack of procedure inlining. However, these overheads can be reduced or eliminated. For example, it is possible to do run-time procedure inlining.

5.3 Latency

Latency was measured for the C and PLAN-P version of the learning bridge as well as for a direct connection between two hosts and the simple user-level repeater. Figure 5 (left) shows the round-trip latency for packets ranging in size from 32 to 4096 bytes measured using the `ping` facility. The difference between the measured latency and the micro-benchmark results are likely due to operating system effects. The large increase observed for packets of 4096 bytes is due to packet fragmentation.

5.4 Throughput

Throughput was measured using `ttcp` with packet sizes varying from 32 to 8192 bytes. Figure 5 (right) shows the resulting throughput for a direct connection, the C repeater, and the C and PLAN-P learning bridge. The throughput measured for the C learning bridge is about the same as the one for the C repeater since the per packet cost of the C learning bridge is very small (4.7 μs). Although the PLAN-P learning bridge incurs some loss of throughput, it still achieves good performance. The average loss of throughput for packets of 256 bytes and larger is 6.3Mbps with respect to the learning bridge written in C. Packets of 1024 bytes have a throughput of 3450 frames per second. In comparison to results on the Pentium architecture presented by Alexander *et al.* [1], we achieve twice as much throughput. The following table compares our results with the results of Alexander.

TTCP 8k bytes packets	Platform	Direct Throughput A	C-Repeater Throughput B	Active bridge C
Alexander-97	Pentium-166	76Mb/s	36.6Mb/s	16Mb/s
			44% of A	43% of B
PLAN-P	Sun Ultra 170	73.7Mb/s	42.9Mb/s	35.6Mb/s
			58% of A	82% of B

6 Related Work

An Active Network node was developed by Wetherall and Tennenhouse [15] based on a TCL interpreter. The emphasis of this work was the validation of the idea.

The most related work to PLAN-P is naturally the PLAN project. We make three important contributions to this work: (1) PLAN is extended to allow application protocols to be expressed (2) safety and security of PLAN programs are strengthened (3) Efficient execution of programs is obtained via run-time specialization.

Java is used to develop protocols for active nodes in the Active Network Transfer System project [16]; a toolkit has been developed for the deployment of protocols. As for any general-purpose language, strong properties like termination cannot be ensured about programs. As a result, it is safer to confine programs written in a language such as Java to the service level rather than using it as a “glue” language.

An alternative to the late program checking approach is proof-carrying code (PCC) [9, 10]. PCC consists of sending together with a program (possibly in binary form) the proof of some property. As a result, a code receiver can accept code from an untrusted producer provided its proof can be validated. This approach has the advantage of allowing binary code to be downloaded as opposed to a higher-level representation of programs, as advocated in this paper. However, it has a number of drawbacks which make its applicability to active networks difficult. The main problem is that the proof associated with the code is only concerned with a specific property; it is not possible to determine a different property or a more strict property given that the source is *a priori* not available or written in a general-purpose language. In fact, one can imagine that, considering the diversity of the network, various sites may have different safety and security requirements for running router programs. In the active network context, the PCC approach seems to require the entire network to standardize the properties router programs need to carry proofs for. This standardization seems both premature and contrary to the motivation of active networks for open-endedness to address rapid evolution of networking needs.

A second problem with PCC related to the previous one is the fact that a proof must be generated by the programmer for each property required by the network nodes. Evolution or change of these properties may result in a need for a new or modified proof.

In contrast to PCC, our approach allows network nodes to perform late program checking for arbitrary properties since the code received is in a high-level form. Yet, this property checking can be a fast process (*e.g.,* syntactic) if the language has been appropriately defined.

Run-time specialization can be seen as an automatic approach to dynamic code generation. Poletto *et al.* propose a manual approach to dynamic code generation [11]. To do so, they extend the C programming language with constructs and types which allow the programmer to directly express the computations which construct program fragments dynamically. The advantage of this strategy is to allow the programmer to have tight control over the program fragments to be constructed at run time. Its main disadvantage is that writing programs that construct programs is a difficult and error prone task. In fact, it is not clear that manual dynamic code generation can scale up to the kind of interpreters required to program networks.

Contrary to manual dynamic code generation, Tempo specializes programs automatically. In particular, the 5000-line PLAN-P interpreter did not require extra operations or declarations to make it amenable to program specialization.

7 Conclusions and Future Work

We have made several extensions to PLAN, an active network language for diagnostics, to allow it to tackle application protocols. This enriched language, called PLAN-P, has been illustrated with two protocols: adaptive compression and active bridging. We have shown how a DSL approach contributes to strengthening safety and security while allowing flexibility. The restricted semantics of PLAN(-P) makes it possible to guarantee that: (1) packets do not cycle within the network (without a run-time resource bound), (2) all packets are delivered, and (3) packet duplication is linear. All these properties can be automatically checked. Finally, we experimentally demonstrated that run-time specialization allows an active network language to be easily implemented by an interpreter while achieving efficiency of compiled programs. More precisely, we showed that a PLAN-P program runs as fast as an equivalent compiled and optimized Java program.

An important result of this work is that the DSL approach combined with interpretation allows late checking of arbitrary properties on active network programs while achieving portability and efficiency.

Although run-specialization has made it possible for PLAN-P programs to run as fast as equivalent Java programs, performance can still be improved. In particular, procedure inlining could be performed during run-time specialization to generate more efficient code. Another topic of interest is the integration of PLAN-P into an operating system kernel in order to improve performance. The DSL approach should ensure this extension will not compromise safety and security. Regarding PLAN-P, we would like to enrich the language to make it expressive enough to implement node services. An advantage of writing services in a DSL is to enable strong properties to also be ensured at this level, and thus to ease the deployment of services on the network. Finally, we are studying various new protocols and applications in the area of multimedia to gain more experience with the language and to gather more experimental data to further validate the advantages of our approach.

References

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

[2] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noyé, S. Thibault, and N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 1998. To appear.

[3] C. Consel, L. Hornof, F. Noël, J. Noyé, and E.N. Volanschi. A uniform approach for compile-time and run-time specialization. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation, International Seminar, Dagstuhl Castle*, number 1110 in LNCS, pages 54–72, February 1996.

[4] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[5] Michael Hicks, Pankaj Kakkar, and Jonathan T Moore. Plan: A programming language for active networks. Technical report, University of Pennsylvania.

[6] G. Muller, R. Marlet, E.N. Volanschi, C. Consel, C. Pu, and A. Goel. Fast, optimized Sun RPC using automatic program specialization. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[7] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, pages 1–20, Portland (Oregon), USA, June 1997. Usenix.

[8] G. Muller, E.N. Volanschi, and R. Marlet. Scaling up partial evaluation for optimizing the Sun commercial RPC protocol. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 116–125. ACM Press, June 1997.

[9] G. Necula. Proof-carrying code. In *Conference Record of the* 24^{th} *Symposium on Principles Of Programming Languages*, pages 106–116, Paris, France, January 1997. ACM Press.

[10] G. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *ACM SIGSOFT Second Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[11] M. Poletto, D. Engler, and M. Frans Kaashoek. `tcc`: A system for fast, flexible, and high-level dynamic code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–121, Las Vegas, Nevada, June 1997. ACM Press.

[12] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, jan 1997.

[13] Scott Thibault and Charles Consel. A framework for application generator design. In *Proceedings of the Symposium on Software Reusability*, Boston, MA, USA, May 1997.

[14] Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[15] D. Wetherall and D. Tennenhouse. The ACTIVE IP Option. In *7th ACM SIGOPS European Workshop*, Connemara, Ireland, 1996.

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

A Proof Outline

While the local processing of each packet is guaranteed to terminate due to the lack of recursion and general loops, remote calls are recursive and thus may not terminate. That is, a packet may cycle within the network. However, due to the domain specific nature of the PLAN(-P) language it is possible to statically and automatically prove sufficient conditions for global termination.

In order to prove termination, it is sufficient to show that some function of the program state, whose domain is a well-founded set, decreases at each recursive call. In PLAN-P, the recursive calls are the `OnNeighbor` and `OnRemote` primitives. The `OnNeighbor` primitive accepts the next node as an argument and we require that this value is obtained via a built-in routing primitive (e.g. `defaultRoute()`) whose argument is the final destination. The `OnRemote` primitive accepts as an argument the final destination and uses the built-in `defaultRoute` primitive to compute the next node. We assume that the built-in routing functions return non-cyclic paths for all possible destinations on all nodes.

We chose the function of our state to be

$$g(\sigma) = (|\mathrm{Net}| - |\sigma(destinations)|, |\mathrm{Net}| - |\sigma(path)|)$$

where σ is the state, Net is the set of all nodes in the network, $\sigma(destinations)$ is the set of all destinations used thus far, and $\sigma(path)$ is the set of all nodes visited towards the current destination. The domain is the well-founded set Nat^2 with lexicographic ordering. At each recursive call, this function will decrease if either:

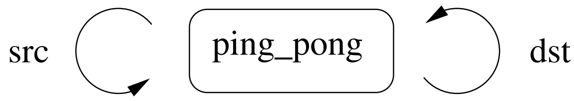

Figure 6: Abstract model of ping_pong example

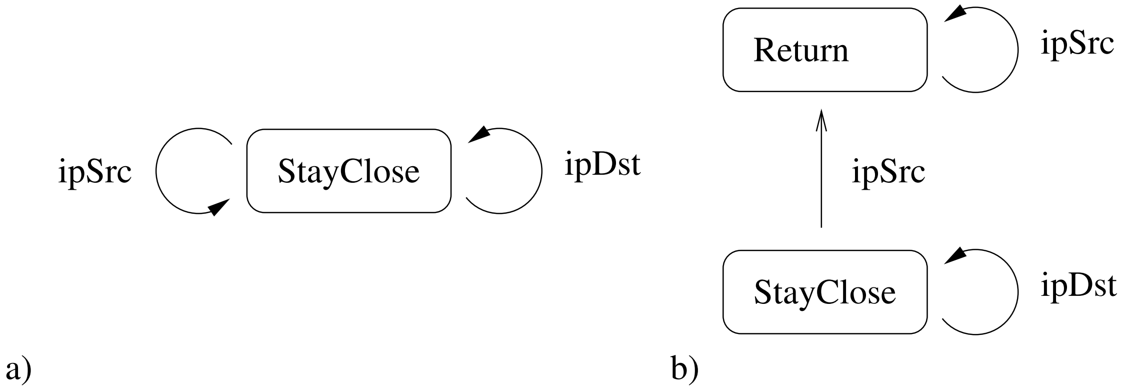

Figure 7: Abstract model of StayClose a) version 1, b) version 2

1. The destination is the same as the previous recursive call, in which case our assumption on the routing functions guarantees the next node is not in $\sigma(path)$.
2. The destination is not a member of $\sigma(destinations)$.

This is equivalent to proving that

$$nc = (destination = last\_destination) \vee (next\_node \notin path)$$

is invariant at each recursive call. We verify *nc* on a given program using exhaustive techniques on an abstract model of the program.

The abstract model of the PLAN-P program is a state machine whose nodes correspond to the channels in the program and whose transitions correspond to `OnNeighbor` and `OnRemote` primitives. The set of host values that could be used as a destination is determined for each call to these primitives by a symbolic fix-point on the program. For each possible destination at each call we add a transition to the state machine to model the call. The proof of *nc* is then deduced via exhaustive state-space exploration. Since the state exploration is performed on an abstract model of the program, its cost is small.

As an example, consider the following PLAN program which does not terminate.

```
fun ping_pong(src : host, dst : host) : unit =
  if thisHost() <> dst then
    OnRemote(ping_pong(src,dst),dst,getRB(),defaultRoute)
  else
    OnRemote(ping_pong(src,dst),src,getRB(),defaultRoute)
```

The symbolic fix-point reveals that the first call to OnRemote could only forward the packet to `dst`, whereas the second call could only forward the packet to `src`. The resulting abstract model is depicted in figure 6, which clearly does not satisfy *nc*.

Although we prove sufficient but not necessary conditions, programs can be easily written so that they meet the sufficient conditions. The main approximation of our analysis is the use of an abstract model which ignores properties of the program state. This abstract model is necessary to obtain a reasonably sized state space. The following example illustrates a program which always terminates, but cannot be proved using our abstract model because it relies on a property of the program state.

```
channel StayClose(ps : unit, cs : unit, p : ip*int*blob) =
  let
    val hop=(#2 p);
    val new_p=(#1 p,hop+1,#3 p)
  in
    if (hop<100) then
      OnRemote(StayClose,ipDst(#1 p),new_p)
    else
      OnRemote(StayClose,ipSrc(#1 p),new_p)
  end
```

The example prevents packets from traveling more than 100 hops. The abstract model for this example, given in figure 7a, is the same as for ping_pong and clearly cannot be shown to terminate. However, by further inspection, it is clear that the example does terminate because the value of hop is monotonicly increasing. Using this knowledge, we can express the fact explicitly in the program as follows.

```
channel Return(ps : unit, cs : unit, p : ip*int*blob) =
  OnRemote(StayClose,ipSrc(#1 p),(#1 p,(#2 p)+1,#3 p))

channel StayClose(ps : unit, cs : unit, p : ip*int*blob) =
  let
    val hop=(#2 p);
    val new_p=(#1 p,hop+1,#3 p)
  in
    if (hop<100) then
      OnRemote(StayClose,ipDst(#1 p),new_p)
    else
      OnRemote(Return,ipSrc(#1 p),new_p)
  end
```

Now, the corresponding abstract machine, shown in figure 7b, is easily verified to satisfy *nc*.

B PLAN-P Grammar

```
planp ::= "protocol" name [init-expr] "is" defs;
init-expr ::= "initstate" expr;
defs ::= {def};
def ::= exndef | fundef | valdef | channeldef
exndef ::= "exception" name
fundef ::= "fun" name parameters ":" type-expr "=" expr
channeldef ::= "channel" name parameters [init-expr] "is" expr
valdef ::= "val" name ":" type-expr "=" expr
parameters ::= "()" | "(" name ":" type-expr {"," name ":" type-expr} ")"

type-expr ::= type-term {"*" type-term}
type-term ::= type-primary [unary-constructor]
unary-constructor ::= "list" | "hash"
type-primary ::= base-type | paren-type
base-type ::= "unit" | "int" | "char" | "string" | "bool" | "blob" | "host" |
              "etherhost" | "interface" | "ethernet" | "ip" | "udp" | "tcp"
paren-type ::= "(" type-expr ")"

expr-list ::= expr {"," expr}
expr ::= simple-expr {binary-op simple-expr}
binary-op ::= "+" | "-" | "*" | "/" | "and" | "or" | "::" |
              "=" | "<" | ">" | "<=" | ">="

simple-expr ::= avalue | op-expr | if-expr | tuple-expr | try-expr | raise-expr |
                let-expr | sequence-expr

if-expr ::= "if" expr "then" expr "else" expr
tuple-expr ::= "()" | "(" expr-list ")"
try-expr ::= "try" expr "handle" name "=>" expr;
raise-expr ::= "raise" name;
sequence-expr ::= "(" expr {";" expr} ")"
list-value ::= "[]" | "[" expr-list "]"

let-expr ::= "let" {letdef} "in" expr "end";
letdef ::= exndef | valdef

op-expr ::= call-expr | unary-op expr | nary-expr
unary-op ::= "not" | "hd" | "tl" | "fst" | "snd" | "#" integer

call-expr ::= name tuple-expr

nary-expr ::= foldl-expr | foldr-expr | print-expr
foldl-expr ::= "foldl" "(" name "," expr "," expr ")"
foldr-expr ::= "foldr" "(" name "," expr "," expr ")"
print-expr ::= "print" "(" expr ")"

avalue ::= name | "true" | "false" | list-value | integer | character | string |
           host-value
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399