Safe and Efficient Active Network Programming

Scott Thibault, Charles Consel, and Gilles Muller
IRISA / INRIA - Université de Rennes 1
{sthibaul,consel,muller}@irisa.fr
http://www.irisa.fr/compose*

Abstract

Active networks are aimed at incorporating programmability into the network to achieve extensibility. One approach to obtaining extensibility is to download router programs into network nodes. This programmability is critical to allow multipoint distributed systems to adapt to network conditions and individual clients' needs. Although promising, this approach raises critical issues such as safety to achieve reliability despite the complexity of a distributed system, security to protect shared resources, and efficiency to maximize usage of bandwidth.

This paper proposes the use of a domain-specific language, PLAN-P, to address all of the above issues. To address safety and security, we give examples of properties of PLAN-P programs that can be automatically *checked due to the use of a restricted language. For efficiency, we show that an automatically generated* run-time *compiler for PLAN-P produces code which outperforms an equivalent compiled Java program. Additionally, we present performance results on a real application (learning bridge) where we obtain 100% of the maximum possible throughput.*

1. Introduction

Multipoint applications are becoming increasingly more important in distributed systems. These applications include multimedia applications and such as audio/video distribution, and Web applications such as intelligent Web caches/agents. Due to varying network conditions these applications must adapt to their environment. Where classical client-server adaptation strategies can be implemented at the end points with good service quality, end-point implementations lead to poor quality of service for multipoint applications. By allowing an application to download programs into routers, *active networks* permit the application to control the processing of packets [13]. Multiple strategies can, thus, be implemented at the router level and decisions to adapt can be made based on the local router environment. For example, data packets might be compressed, encrypted, or dropped depending on the network conditions and receiving application needs.

The introduction of highly-programmable networks, however, raises a number of issues [13]. First, safety because distributed systems are complex and difficult to debug. Second, security should also be ensured because network routers are shared resources. Third, given the heterogeneity of the network, the module executing router programs should be highly portable. Finally, the execution of router programs must be efficient in order to maximize bandwidth and not degrade the bandwidth of normal traffic.

Java has been purposed as a solution to heterogeneity based on bytecode interpretation. It also provides some level of safety and security. For example, Java provides type safety and guarantees non-interference of concurrent processes. However, in order to remain general purpose, the level of safety and security that can be achieved is limited. For example, termination of Java programs is undecidable.

Domain-specific languages (DSL) are languages that are, among other things, suitably restricted to enable strict constraints to be enforced and specific properties to be determined [6]. For example, the absence of recursion and general loops can guarantee termination. Previous work on DSLs and active networks suggests that the DSL approach is a promising direction [5, 12]. In this paper, we propose a safe and efficient approach to programming application protocols based on a DSL: PLAN-P. More precisely it offers the following three contributions.

Safety and security. As advocated by promoters of DSLs, because of their simple semantics and restrictions, such languages make it possible to determine properties typically undecidable in general-purpose languages (*e.g.,* termination). PLAN-P programs can automatically be checked 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

*This work has been partly supported by FRANCE TELECOM contract CNET 96-1B-027.

these properties can be automatically checked due to certain restrictions or domain-specific attributes of the language which would otherwise make automatic proof impossible.

PLAN-P: A programming language for application protocols. Previously described DSLs used in active networking [5, 12] mostly target network diagnostic applications. We have extended the PLAN [5] 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.

High-performance PLAN-P execution via run-time specialization. It is too costly to build routers with the fastest processors available, so efficiency must be obtainable on lower-end processors. The third contribution of our work is in improving the performance of PLAN-P programs, and more generally programs which require *late checking*. That is, the program source must be sent to the routers to allow them to do defensive checking at run time for safety and security reasons. Sending more efficient, lower-level, representations makes defensive checking difficult, if not impossible. Downloading the source requires that PLAN-P 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-P. However, regardless of the compilation technology, an additional 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 program specialization [2]. Program specialization is a program transformation that specializes a program with respect to some known inputs by computing expressions that only depend on these inputs. We have implemented a PLAN-P interpreter which is written in C to be portable on all routers. However, unlike the existing implementations of active network DSLs, our interpreter is specialized with respect to a PLAN-P program in order to eliminate the interpretive overhead. Because the program is not available until run time, a run-time form of program specialization is used. Our approach relies on a program specializer for C called Tempo which is able to specialize programs at run time [3].

As discussed in section 5, our experiments show that the PLAN-P interpreter specialized with respect to a learning bridge program achieves up to 100% of the throughput obtained with an equivalent bridge written in C. More importantly, we have implemented a bridge in Java to assess the performance of a highly-portable language. We compiled this program with an optimizing off-line Java compiler [8] to get the best possible performance. When compared to our specialized PLAN-P interpreter, the specialized program outperforms the Java program by a factor of 2 in a micro-benchmark. In comparison to results on the Pentium architecture presented by Alexander *et al.* [1] using CAML bytecode, we achieve 2.5 times 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 PLAN-P. Interpreters also permit prototyping and are easier to develop and maintain.

The rest of this paper is organized as follows. Section 2 presents the language PLAN-P and describes two example protocols. Section 3 describes three properties that can be automatically proved for PLAN-P programs. Section 4 presents an efficient implementation of PLAN-P based on program specialization. Section 5 describes an experiment aimed at evaluating the performance of the PLAN-P system. Finally, we present related work and conclusions in sections 6 and 7 respectively.

2. The PLAN-P Language

PLAN-P is a domain-specific language for active network application protocols. It is an extension of PLAN, a previously developed DSL for active networks [5]. The PLAN 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

A main limitation of PLAN is that it is not sufficiently expressive to specify application protocols. In order to specify these protocols, we have extended PLAN with 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

```
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) < capacityThreshold then
        (OnNeighbor(compressed,next,(fst(p),compress(snd(p)))); (ps,cs))
      else
        (OnNeighbor(uncompressed,next,p); (ps,cs))
    end
  else
    (deliver(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((fst(p),uncompress(snd(p)))); (ps,cs))
```

Figure 1. Adaptive compression protocol

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.

- 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 with each packet which provides a state-transition view of the global behavior of a packet traveling through the network.

Additionally, our extensions permit us to be compatible with existing protocols such as IP and TCP, to modify the behavior of these existing protocols, to introduce new packet types and behavior for new protocols, and to introduce session-specific packet types and behavior. Due to space limitations, a more detailed language description is available as a technical report [15]. The following two sections illustrate the language by example.

2.2. Adaptive Compression Example

Figure 1 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. For example, the compression protocol defines a new protocol built on IP as indicated by the declared packet type (`ip*blob`).

When multiple channels are defined, an additional state is implicitly associated to packets that indicates which channel the packet belongs to. 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 [17]. 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 (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.

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

The packet must then be uncompressed before it is delivered.

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 2 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.

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 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 in PLAN-P

Adding programmability to a distributed 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 [13]. 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-P is an example that demonstrates how a DSL 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 for PLAN-P programs.

Global termination. First we consider the termination of PLAN-P programs. PLAN-P has the property that execution of a 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.

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 destina-

```
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 2. Extract of learning bridge

tion. 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 these built-in routing functions return non-cyclic paths for all possible destinations on all nodes.

If we can show that

$$nc = (this\_dest = last\_dest) \vee (next\_node \notin path)$$

is invariant at each recursive call then we can show that the function

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

decreases at each recursive call, 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.

We verify nc for 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.

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 underlying 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 (or delivered) 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.

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.

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

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 [7]. In fact, the specialization costs for the PLAN-P interpreter only require the specialized program to be executed a few times to be amortized (*i.e.*, it takes less time to specialize the interpreter and execute the result a few times than it does to interpret the program that many times).

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, 7]. It has been successfully used for various applications ranging from system 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 [7]. Tempo operates on both the SPARC and Pentium architectures under Solaris and Linux[3].

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.

4.3. Specializing the PLAN-P interpreter

The structure of the interpreter for PLAN-P follows the methodology defined by Thibault and Consel [14]. 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 [16].

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 50 times faster than the interpreter clearly demonstrating the high degree of specialization of the PLAN-P interpreter.

[3]Tempo is publicly available and can be downloaded from `http://www.irisa.fr/compose/tempo`

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 learning bridge that was described by Alexander *et al.* [1] and presented in section 2.3. The learning algorithm uses a hash table to record the source address and the LAN of received packets[4], 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.

5.1. Performance measurements

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.

Performance measurements are presented for latency and throughput for several programs both in supervisor and user mode. Results are given in user mode in order to compare with the JVM bytecode interpretation, which could not be integrated into the kernel. To show the maximum performance possible with the given configuration, the performance of a direct connection between two machines is given. In order to measure the cost of running in user mode, we present results for a minimal bridge in C that simply receives packets and sends them on the other interfaces. This C repeater serves as a base line. The remaining measurements are for various implementations of the learning bridge presented above. These implementations include a hand-written version in C compiled with gcc, a hand-written version in Java compiled to Java bytecode, and a version in PLAN-P. Measurements are given for the PLAN-P bridge using both interpretation and run-time specialization.

Round-trip latency for packets ranging in size from 32 to 4096 bytes was measured using the `ping` facility. Throughput was measured using `ttcp` with packet sizes varying from 32 to 8192 bytes. Figure 3 shows the throughput results using TCP in user mode. Table 1 summarizes these results and the results obtained using UDP in supervisor mode.

5.2. Performance discussion.

The performance ordering of the various programs shown in figure 3 are as one would expect. The PLAN-P interpreter, implemented as a standard abstract syntax traversal, is the slowest. The JVM is a bytecode interpreter which is a simplified form of interpreter, typically implemented in a single loop. Thus, the JVM is faster than a full-language interpreter, but still has considerable interpretation overhead. Next, as expected, the specialized PLAN-P interpreter eliminates the interpretation layer of the program and is thus faster than both full interpretation or bytecode interpretation. Finally, the hand-written C code is slightly faster than the code automatically produced by specialization.

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

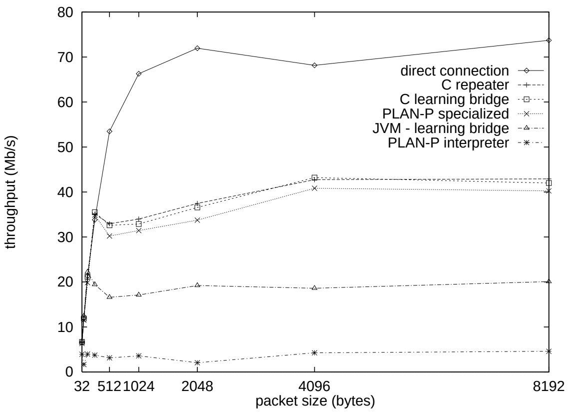

Figure 3. Throughput

TTCP	UDP		TCP	
8kb packets	Throughput	Perc.	Throughput	Perc.
C repeater	88.1 Mb/s	100%	42.9 Mb/s	100%
C learning	88.1 Mb/s	100%	42.0 Mb/s	98%
PLAN-P Specialized	88.1 Mb/s	100%	40.2 Mb/s	94%
JAVA JVM			20.1 Mb/s	47%
PLAN-P Interpreted			4.6 Mb/s	11%

Table 1. Performance summary

User-mode/TCP performance. Table 1 shows that full-language interpretation is clearly not a viable approach. Using bytecode interpretation obtains substantial improvement, but is not sufficient to make the approach practical. Alexander also reports similar results using CAML bytecode interpretation where he obtains up to 43% of the maximum [1]. Finally, we find that the throughput obtained with the specialized version is very good, only 4% less than a hand-written C program.

Another question to ask might be, how does specialization compare to JIT technology? To answer this question we compiled the Java learning bridge using Harissa, an off-line optimizing Java compiler, which generates considerably better code than existing JIT compilers [8]. On a micro-benchmark, the compiled Java code is twice as slow as the specialized PLAN-P interpreter. In terms of throughput, the compiled Java version can achieve 85% of the maximum. Although 85% is acceptable, we obtain better performance with specialization and can prove much stronger properties about the safety and security of PLAN-P programs.

Supervisor-mode/UDP performance. Although we could not integrate the Java versions into the kernel for comparison, we are still interested in the performance of PLAN-P, which is intended to operate within the kernel. We repeated the bridge experiment with the PLAN-P kernel module and the UDP protocol instead of TCP. In this case, the specialized PLAN-P interpreter achieves not only 100% of the throughput of the hand-written C bridge for large packets, but 100% of the maximum bandwidth of the network (*i.e.*, the bandwidth of a direct connection). This is because processing a single packet takes less time than it does to physically transmit the packet. In order to discover the difference in computation time between the hand-written C program and the specialized PLAN-P version, we measured the per packet processing time within the kernel using hardware counters. These measurements show that the specialized program is only 58% slower than the hand-written C program in terms of cycles per packet.

6. Related Work

An Active Network node was developed by Wetherall and Tennenhouse [17] based on a TCL interpreter. The emphasis of that 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 [18]; 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 presented PLAN-P, an active network language for application protocols. We have illustrated PLAN-P with two protocols: adaptive compression and active bridging. We have shown how the 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 faster than an equivalent compiled and optimized Java program and achieves equivalent performance to a hand-written C program. An important result of this work is that the DSL approach combined with specialization allows late checking of arbitrary properties on active network programs while achieving portability and efficiency.

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 to ease service deployment 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.

8. Acknowledgements

Pierrick Gachet helped us to setup experiments and provided technical assistance on the Solaris kernel. We wish to thank Renaud Marlet for his comments on a previous draft of this paper and his ideas for improving the run-time specializer.

References

[1] D. S. Alexander, M. Shaw, S. M. Nettles, and J. 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. 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, Feb. 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, Jan. 1996.

[5] M. Hicks, P. Kakkar, and J. T. Moore. PLAN: A programming language for active networks. Technical report, University of Pennsylvania.

[6] D. A. Ladd and J. C. Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Fourth International World Wide Web Conference*, Boston, Massachusetts, Dec. 1995.

[7] G. Muller, R. Marlet, E. 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.

[8] 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.

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

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

[11] M. Poletto, D. Engler, and M. F. 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] B. Schwartz, W. Zhou, A. W. Jackson, W. T. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks. BBN Technologies, Jan. 1998.

[13] 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.

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

[15] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. Publication interne 1170, IRISA, Rennes, France, Jan. 1998.

[16] S. Thibault, R. Marlet, and C. 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, Oct. 1997.

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

[18] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH*, Apr. 1998.