AAN : A Highly Reflective Active Network*

CHRISTIAN KHOURY
Université Paris 6, LIP6/CNRS
Paris, France
email: Christian.Khoury@lip6.fr

BERTIL FOLLIOT
Université Paris 6, LIP6/CNRS
Paris, France
email: Bertil.Folliot@lip6.fr

IAN PIUMARTA
Université Paris 6, LIP6/CNRS
Paris, France
also at INRIA Rocquencourt, SOR Project
Rocquencourt, France
email: Ian.Piumarta@{lip6,inria}.fr

Abstract

Active Networking strive to make network evolution quicker and easier by adding extensibility to the latter. However, extensibility of current prototypes rely mostly on the language used for their implementation, somehow blocking the march towards a fully flexible network. In this paper, we present the architecture and implementation of a highly extensible active network node, *AAN*, based on a reflective and specialize-able virtual machine. Such a platform enables building active networks and applications, fitting exactly their resource consumption needs. ANTS and PLAN, two of the first active networks' prototypes, have already been implemented on this platform as a concept of proof.

KEY WORDS

Active Networks, Architectures, Extensibility, Flexibility, Specialize-able Virtual Machine, Runtime Code Generation

1 Introduction

Nowadays, networks' architectures suffer from evolution problems; IPv6 [1] has been defined for years now without any glimpse of hope of finally replacing IPv4 [2]. Active Networks [3] seek to change this situation by making the network's nodes programmable, therefore making it possible to define new protocols on the fly. A interesting , though perilous, side effect of this approach is the possibility of opening the internal network's resources to applications' use.

Current prototypes offer limited extensibility as they depend upon the facilities offered by the language used for their implementation; PLAN [4] and ANTS [5], one of the early active networking platforms, use the extension features of respectively Java [6, 7] and Objective Caml [8]. Active Networks are also concerned with all the issues tackled in mobile code environments : portability, safety, security, code compactness and performance. Some of the latters are partly in conflict as performance brings less flexibility and less portability, hence less compactness[1]. Others, e.g. safety, are highly requested in some environments, e.g. open networks, and superfluous or even penalizing in others, e.g. already secure networks.

Many active networking platforms (ANs) have emerged, each proposing an architecture based on its targeted applications' specific needs, and reconciling at best some of the reviewed properties. For example, some aim at network management [9], while others aspire to running general services; this lead to the definition of a multitude of overlapping **fixed** abstractions in existing ANs, hence making the choice of the programmer a bit tricky and mostly based on the knowledge of the programming language more than on the AN platform itself. Moreover, some of these abstractions may be of no interest to some applications, and their unwanted presence may affect performance adversely.

In this paper, we present the conception and evaluation of a highly extensible active network node, the Active Active Network (*AAN*), designed to reconcile all the properties cited above, based on a specialize-able virtual machine coupled with a high level reflective language. The first *Active* refers to the fact that the active node itself is re-definable.

The remainder of this paper is organized as follows. Section 2 gives a view of related work. Section 3 details the architecture and implementation of AAN. Section 4 presents preliminary evaluation of the platform, and section 5 presents our conclusions and directions for future research.

2 Related Work

A number of architectures have been proposed and developed to test active networks or make the network more flexible. In this section, we briefly review some of the most no-

[1] compiled vs interpreted languages

table realizations, their key concepts, as well as their characteristics.

The *Click Modular Router* [10] is a software architecture building flexible and configurable routers; it provides elements (packet processing modules) which can be composed together, described in a directed graph, to create the desired routing node. It is written in *C++*, which makes it quite efficient, but lacks extensibility features on the fly. Besides, it does not change the packet headers; the routers built are just more easy and quicker to build.

Smart Packets [9] tackle the issue of network management. For programming the packets, two languages have been devised : *Sprocket*, a light and safer version of *C*, and Spanner, a bytecode created to make packets more compact. Both languages run on the same virtual machine which interprets Spanner bytecode. Safety is based on certification : each packet carries a key authentifying the sender. Therefore, only known users get to use the node.

ANTS [5] is a general purpose active network. It is built on top of the Java virtual machine. Packets contain references to the code to be executed. Upon reception, a node looks for the referenced code, and, if it is not already loaded, it queries it from the last sender. The node's extensibility is *restricted* by Java's class loader facility.

PLANet [4], is another general purpose active network. It is based on a two-layer architecture : the lower layer offering certified safe services written in Objective Caml, thus mainly static. The upper layer defines a language, *PLAN*, with safety properties, e.g. termination, used for transporting code over the network. Extensibility is once again based on the Objective Caml features; as the latter's bytecode is not safe, extensibility at run-time is not recommended.

All the architectures reviewed above give their own perception of what should be an active network node's architecture; they generally define static abstractions meeting the needs of their targeted applications. However, often, it seems interesting to mix features from different architectures or to alleviate some node from needless services. Such a task is impossible should we use any of the existing platforms, as they rest on some *fixed set* of abstractions.

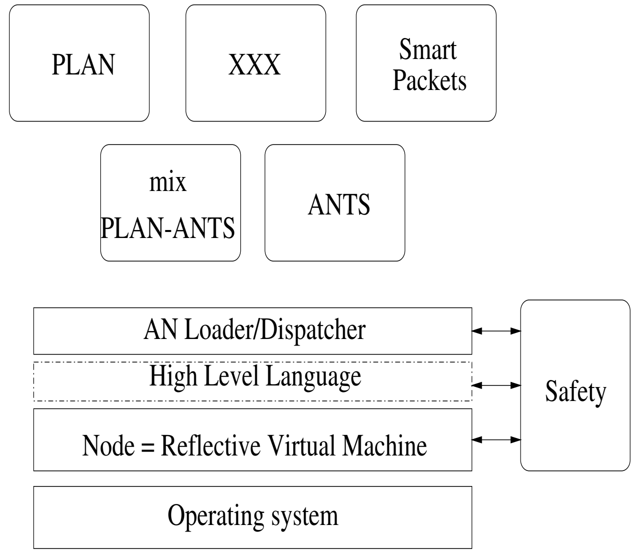

Figure 1. Active Active Network architecture

```
(define func (lambda() (+ 2 3)))

(func)

; in bytecode format
push0
pushGlobal   0 = func
apply
```

Figure 2. Applying a normal function

3 Active Active Network

In this section, we present the architecture of our Active Active Network (see figure 1), *AAN*. It is a step forward to even more flexibility than existing architectures as it brings extensibility at the active network's level, i.e. the active network itself can be changed, replaced, … In the sequel, we will briefly describe each component of this architecture.

One of the main requirements of an active node is *portability*; it is best achieved using a VM-based approach, hence the use of a VM in AAN. To fit all possible needs, a VM has to be minimal and highly specialize-able; the *Reflective Virtual Machine* (RVM) is similar to a traditional virtual machine except that it can rewrite its own instruction set (specialize itself) at *run time* according to any specific applications' needs. It is a highly reflective virtual machine, with minimal facilities; all of the objects representing executable and execution state (functions, closures, function activations, and so on) are both first-class and directly accessible to the programmer. The most important consequences of this are that high level facilities (continuations and processes, for example) are straightforward to implement as user-level code *whenever needed*. The first step in creating an extension to the RVM is to translate the specification (a user function) into *native code*, using runtime code generation, and assigning to it a new *bytecode*, using the *define-instruction* primitive. Exporting an extension in the form of a primitive boosts performance[2], and has the advantage of reducing the application's code size, since the overheads associated with normal function application are no longer incurred (see figures 2 and 3).

On top of this common specialize-able medium, i.e. the RVM, we need the most basic construct for specializing a node, that is a *VM loader*. A crucial issue for a loader is to know *what* to load in order to process each packet received. There exists two possible ways of handling this

[2] Primitivized functions run, on average, 4 times faster than non primitivized ones

```
(define-instruction 'newbytecode func)

(newbytecode)

; in bytecode format
udi ; user defined instruction
```

Figure 3. Applying a new bytecode

issue : (1) use a type[3] inside the packet to identify the VM modules needed for processing it, or (2) consider the packet as being an executable capable of specializing the node and processing itself. Hereafter, we describe these two possible loaders and their prerequisites. The code for the first loader is shown in figure 4.

```
(define loader
  (lambda ()
    ...
    ...        ;; some definitions
    ...
    (while t   ;; infinite loop
      (setq pkt (recv-pkt))
      (setq exec (loadVM
                   (get pkt 0)))
      ;; apply this function to the
      ;; rest of the packet
      (exec pkt))))
```

Figure 4. Code for the first loader

The phases after receiving a packet are as follows :

1. extract the *type* information from the packet,
2. if the VM is already loaded, go to step 3. Otherwise, load the VM corresponding to the AN; all the functionalities are loaded and the execution function of this AN, which knows how to handle the rest of the packet, is returned,
3. the execution function is handed the rest of the packet.

Using such a loader, there must exist a default location protocol at least for the first packet received related to a specific service. Afterwards, if the service creates node-resident state, and defines another location protocol, the latter will be used instead for the next packet received belonging to the same service.

The code for the second loader is shown in figure 5.

In this case, it is up to the loaded packet code to locate and instantiate the AN virtual machine it needs for running.

It is worth noting that in both cases, the loader itself is replaceable- by someone, e.g. a system administrator,

[3] code signature, e.g. MD5 [11]

```
(define loader
  (lambda ()
    ...
    ...        ;; some definitions
    ...
    (while t   ;; infinite loop
      (setq pkt (recv-pkt))
      (exec pkt))))
```

Figure 5. Code for the second loader

or something, e.g. a packet, having the suitable authority-, just by overriding the *loader* function. *Safety*, although not addressed in this paper, can be achieved using module thinning [12, 13], proof carrying code [14] and software-based isolation [15].

On top of these bricks, we defined a small language to facilitate the creation of new virtual machine definitions (see example in figure 6).

```
(defVM newVM 1234567887654321
  [RESOURCES
      (stack (array))
      (memory (array))
      (sp 0)
              (rtbl (...))
      ...
             ]

  [INSTRUCTIONS
              (push   (lambda(x) (ap-
pend stack x)))
              (pop    (lambda() (rm-
last stack)))
              ...
                ]

  [EXECLOOP (lambda () (...)])
```

Figure 6. A virtual machine's description

This language defines a virtual machine by describing semantically its `RESOURCES`, such as its memory model, a routing table, ... its instruction set (`INSTRUCTION`), and its execution model (`EXECLOOP`). Each definition of a virtual machine is given a symbolic name and an identifier (`defVM newVM 1234567887654321`), the latter being typically a code fingerprint. Using this small language, we mimicked two existing active networks, PLAN and ANTS.

4 Preliminary Evaluation

In this section, we present a succinct evaluation of AAN; the measurements were done on a Pentium III, 550 MHz, 128 Kb L1 cache, and 512 Kb L2 cache.

The RVM, as a prototype, performs acceptably compared to other virtual machines; performance results for a *Fibonacci* function written in different programming languages are shown in table 1.

Language	% C
Perl 5.005_03	0.6
Python 1.5.2	1.2
RVM	4.4
RVM Primitivized	14.4
Java	12.8
Java (IBM Jit)	81.4

Table 1. Performance compared to C

We also measured the throughput achieved by our AAN prototype, as shown in figure 7, using an IP-forwarding service; packets contain data and are forwarded as soon as they are received. We see that our prototype forwards approximately between 250 to 300 packets per second, 6 times less than ANTS. This can be explained by the fact that the latter is built using the IBM Jit for Java [16], and the RVM still being in its first release.

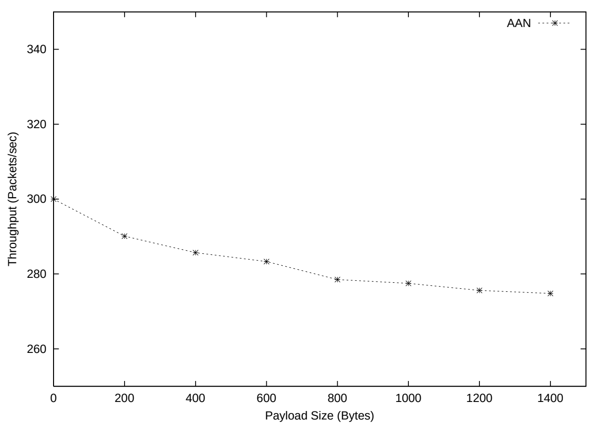

Figure 7. AAN throughput

5 Conclusions

Current active networks often support needless features; besides, their extensibility is limited as it is based on the language used for their implementation. The solution we propose in this paper is based on the simple principle that a minimal extension facility at the lowest level makes possible all kinds of extensions at any higher level; this principle is advocated in the design of new operating systems, such as Exokernel [17] and Think [18]. At the lowest level, we use a highly reflective virtual machine supporting extensibility at its bytecode level. Atop the RVM, we defined the utmost minimal node kernel : a loader. We presented two possible forms of loaders along with their characteristics. As a concept of proof, two existing active networks have been mimicked on this platform. Preliminary measurements were presented showing encouraging figures in favor of using this platform for more in-depth studies, while acknowledging the immaturity of this first release.

References

[1] S. Deering and R. Hinden. Internet protocol, 1998.

[2] J. Postel. Internet control message protocol, 1981.

[3] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[4] Michael W. Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[5] David Wetherall. Active network vision and reality: lessions form a capsule-based system. In *Symposium on Operating Systems Principles*, pages 64–79, 1999.

[6] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1997.

[7] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 1998.

[8] X. Leroy, D. Remy, and P. Weis. Objective caml—a general purpose high-level programming language, 1999.

[9] B. Schwartz, W. Zhou, A. Jackson, W. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks, 1998.

[10] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[11] Ronald L. Rivest. The MD5 message digest algorithm. In A. J. Menezes and S. A. Vanstone, editors, *Proc. CRYPTO 90*, pages 303–311. Springer, 1991.

[12] Xavier Leroy and François Rouaix. Security properties of typed applets. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 391–403, New York, NY, 1998.

[13] François Rouaix. A Web navigator with applets in Caml. *Computer Networks and ISDN Systems*, 28(7–11):1365–1371, 1996.

[14] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.

[15] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[16] V. Seshadri. IBM high performance compiler for java, 1997.

[17] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[18] J-P. Fassino and J-B. Stéfani. Think: un noyau d'infrastructure répartie adaptable. In *Actes de la 2ème Conférence Française en Systèmes d'Exploitation*, 2000.