An Active Platform as Middleware for Services and Communities Discovery

Sylvain Martin* and Guy Leduc

Research Unit in Networking, Université de Liège,
Institut Montefiore B28, 4000 Liège 1, Belgium
{martin, leduc}@run.montefiore.ulg.ac.be
http://www.run.montefiore.ulg.ac.be/

Abstract. In an increasing number of cases, network hosts need to locate a machine based on its *role* in a service or community rather than based on a well-known address. We propose and evaluate WASP, a lightweight active platform where ephemeral state left in the network can help locate service providers such as request dispatchers or computation aggregators. In an active grid architecture, WASP can also help locate participants, build and manage overlays.

1 Introduction

In peer-to-peer systems in general and grid computing as a specific case, we are facing the problem of communities of machines that need to cooperate together without having any *a priori* knowledge of their respective existence. Existing schemes often rely on the existence of some centralizing machine that swaps peers addresses. With the increase of service popularity and the decrease of average connection time, keeping these so-called *pong servers* scalable is a real challenge.

In this paper, we study a solution based on active networks that will help network operators and peer-to-peer application designers to introduce aggregators and caches of *pong servers* transparently in the network. Our active platform also allows discovery of more generic services and could be used to locate computation dispatchers and aggregators as well.

1.1 Why Yet Another Active Platform?

Despite an important amount of active network platforms has been proposed and studied, none has really successfully reached the real-world deployment level. In response to this reality, WASP (*Weightless Active packets for ephemeral State Processing*) is made lightweight enough and resource-friendly so that its presence in the network does not become a nuisance for network managers. WASP is not built from scratch: it is an effort to merge the advantage of two recently proposed platforms ESP (*Ephemeral State Processing*) and SNAP (*Safe and Nimble Active Packets*) [5]) that also focus on

* Sylvain Martin is a Research Fellow of the Belgian National Fund for Scientific Research (FNRS)

safe, flexible and efficient – in a word, *practical* (cf Moore et *al*. [2]) – active networks. Section 2 shows how both computation time, storage and network bandwidth are kept under low and predictable limits for WASP packets.

Unlike fully-featured active platforms, WASP alone is not able to offer complex services such as flow transcoding or realtime auction though it can help *advertise* and *locate* such services, helping machines that offer such services (hereafter called *service providers*) and machines that use such services (hereafter called *end-systems*) to build and manage adequate *tunnels* and *overlays* in an automated way, thus providing an effective support for active grid architectures. We will show in Sect. 3 how WASP can be used to build such solutions.

The design of our proposed platform is detailed in Sect. 4. In order to evaluate its performance, we translated ESP instructions into WASP code and compared execution timings. The results of this evaluation are presented in Sect. 5

1.2 What Slows Down Active Packets

The speed at which our active platform will be able to process active packets will define the network locations where it can actually be deployed. Our goal is to offer a solution that can be running even on a border router in a transit domain, taking advantage of *network processor* technologies to achieve high throughput. A simple *active packet* crossing an active node will incur different types of time-consuming operations:

Classification Filters should be applied so that the active packet is recognized as such. Depending on the architecture, this filtering may be based on a dedicated *transport protocol type*, the presence of an *IP header option*, etc.

Delivery After classification, we still have to make the packet available to the component that will process it. In the worst case, the whole packet should be copied into user-space before applicative decoding could be applied. Most performance-targetted platforms ([3,4]) thus decode and process packets at kernel level.

Decoding Some platforms like ESP [4] and SNAP [5] are able to process the operations for the packet as soon as it is delivered, but in most of the other frameworks, a data decoding phase is needed to deserialize objects, strings, lists, etc.

Processing By interpretation or execution of compiled code from a cache. We chose interpretation since it better fits inherent characteristics of network processors [9].

2 Resource Aware System

2.1 Execution Time

Even if untrusted code cannot perform harmful operations thanks to sandboxing, it is impractical to detect malicious code that will run silly loop, consuming available CPU time on routers without performing any ‘useful’ job.

Like in SNAP[5], WASP avoids this by prohibiting *backward jumps* and providing only instructions with predictable execution time. Together with in-place processing of packets at kernel level, this allows our lightweight control tasks to be performed at the lowest cost for the router.

2.2 Memory and Storage

Most active protocols will need information to be stored temporarily on intermediate nodes, so that it can be later retrieved by other active packets. It is important for network availability and performance that this local storage remains easy to manage and can automatically discard information that is no longer pertinent.

ANTS [1] and many other platforms use *soft-state* based memory management to release memory that has not been used by packets for a given amount of time. Unfortunately, soft-state based managers make it hard for the access control to define if there will be sufficient memory to accept the flow.

Alternatively, ESP [4] proposes the *ephemeral state* approach, where data are kept for a constant period, *regardless of how frequent the data is referenced during that period*. If, in addition, all the data slots in the store have the same size, collecting free-for-reuse slots becomes simple enough to execute without disturbing packet forwarding tasks on the router, and flow access checking simply requires that the router checks how many different slots are used by the flow. We will call *tags* these fixed-size data that the node associates with a key for a fixed amount of time, like in the ESP terminology.

2.3 Network Bandwidth

Taking care of local resources is required to achieve platform *safety* but not sufficient. If no restriction is enforced, an ill-intentioned active packet could easily create clones of itself will all the allowed execution times, and clones of the clones at the next router so that a single emitted packet will overload the destination (in addition of network links close to that destination). In the case of the *WASP* platform, packets do not have the ability to create child packets unless they are targetted at a multicast address on a multicast router. All it can do is block the packet or send it back to its source.

3 Service Discovery with WASP

Literature has presented a number of active network-based solutions to various problems such as real-time auction, hierarchical web caches, video stream filters, reliable multicast and many more. All these applications can be viewed as applying a custom *service* (filtering, merging, splitting of packets) at some strategic points in the network. It is somehow expected that, sooner or later, network operators will integrate such services in their network as their presence could help reduce the required bandwidth. The behaviour of an active service can thus be seen as follows:

1. identify anchor points in the network (*i.e.* machines able to host the service),
2. detect at which anchor point(s) service deployment is strategically most useful,
3. route packets requiring the service towards deployed *service provider(s)* [1],
4. apply merge/split/filter service on packets received by the service provider(s).

In a grid computing environment, those services could for instance consist of collecting available computations sites, routing computations requests towards the closest (or less loaded) point of presence or even perform distribution/aggregation of computation requests hierarchically.

[1] *i.e.* "nodes offering the packet filter/split/merge service", not Internet Service Providers

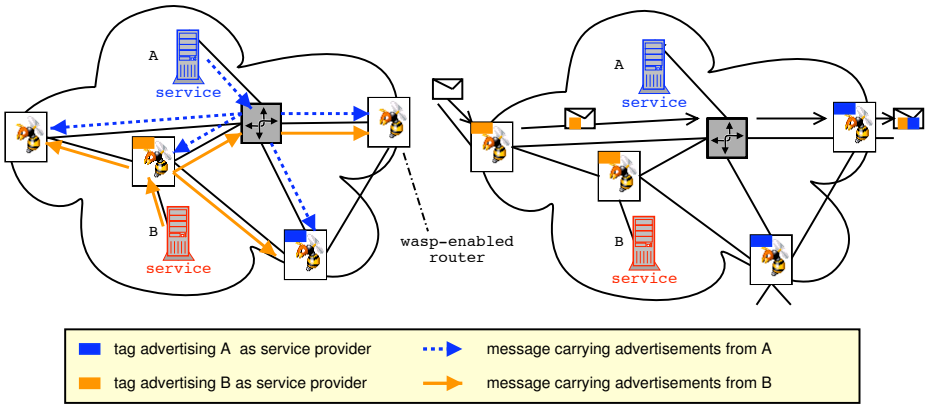

Fig. 1. Advertising (left-side) and looking for (right-side) service

3.1 The Managed Dynamic Overlay Alternative

Most existing solutions for such problems, like OPUS [7] or the X-Bone [8], set up an infrastructure that dynamically creates *overlay networks* interconnecting end-systems and intermediate service providers with *tunnels* in order to obtain the desired logical topology. Unfortunately, none of these works have suggested a truly scalable and completely decentralised method for identifying available anchor points in a very large-scale network. Moreover, maintaining the infrastructure, monitoring the available resources and expressing applications' needs in a generic fashion remains a resource-intensive activity even when hierarchically divided such as in OPUS.

In the following, we will show how, with sensibly less support from all sides, WASP manages to offer enough information to *end-systems* and *service providers* so that they can take strategic decisions themselves.

3.2 Basis of Service discovery with WASP

The idea behind the WASP platform is to provide a lightweight environment that can be used for locating the most interesting service provider(s) independently of what the service will actually do. Once service-providing node(s) have been located, the end-system can adjust application behaviour so that the applicative flow go through the discovered provider(s).

When a new applicative flow is initiated, small active packets are used to probe the network on the route to be taken. Each time such probe crosses a WASP node, it will lookup the node store to see if it can find *advertisements* of the expected service, consisting of the provider address and cost for reaching that provider from the local node. Depending on the application needs, probes will perform some pre-filtering of the collected advertisements or simply store them all. Based on this collected knowledge of the network, the end-system can evaluate the different options and enforce the one that will result in the best utility.

The same kind of active packets can also be used by the service providers to install advertisements in routers of the local domain. Again, the programmability of the advertisement packets allows expression of simple policies such as only keeping the advertisement(s) of the (k) nearest service provider(s) in a router, and report other service providers to the advertiser.

Figure 1 illustrates that two-phase process: servers A and B first flood the domain with WASP packets advertising their presence, avoiding to re-install a tag in a router that already contains a better tag (e.g. advertising a closer or less loaded service provider). A source S can then use another WASP packet to record those tags as a list of provider P and branchpoint X information: $(P_{addr}, X_{addr}, cost(S, X), cost(X, P))$. Note that by simply changing the program in service advertisement and lookup packets, we are able to select the service provider that is closest to the source or to the destination, or to keep only the provider that will lead to smallest path deviation in each domain and leave final selection to the end systems.

3.3 Flooding Locally

In order to advertise the service, providers have to locate WASP routers in the local domain and send them WASP packets that will install advertisement tags. We benefit here from the fact that WASP processing is *optional* so no overlay of WASP-enabled routers need to be pre-established. In our previous work [6], we show how knowing the routing table of the local domain suffices to discover all the active routers of that domain.

A particularity of *ephemeral* storage is that the advertising tag will be deleted after a fixed period τ, regardless of any refresh we could try to perform. Therefore, there may be a small delay between the moment where a WASP router decides to remove an advertisement tag and the moment where an advertisement refresh comes. Even if the server manages to learn precisely the tag's lifetime τ it cannot completely avoid the risk that client packets may not see any advertisement. If this risk cannot be afforded, it is still possible for a service to use two separate keys k_1 and k_2 that will be refreshed with a period $\tau + \epsilon$ but such as advertisements of k_1 and k_2 are separated by a delay of e.g. $\tau/2$. A client that doesn't find the "primary tag" (referenced by k_1) can then check the "backup tag" (referenced by k_2) to see whether the service is really missing.

4 The WASP Platform

As we previously said, WASP is derived from ESP router[4]. An ESP node consists of *Ephemeral State Stores* (ESS) containing *tags* that packets access based on 64 bit *keys*. Each packet requests the execution of one of the pre-defined operations on certain tags. Despite operations can be modified a bit (e.g. changing threshold values, selecting operators, etc), they remain thighly bound to multicast-related applications. It is also a bit disappointing to see that as soon as one wishes to implement more complex feature such as *reliable* flows merging, other very-specific operations need to be added.

The WASP platform thus keeps the overall design of ESP but replaces pre-defined operations by a *virtual processor* interpreting a bytecode language inspirated by SNAP

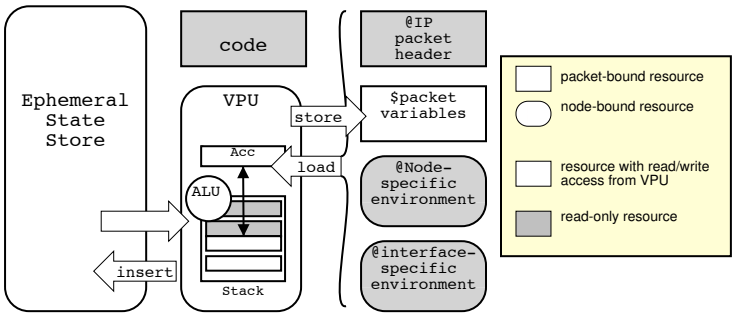

Fig. 2. WASP Execution Environment

[5]. Another couple of extensions have been brought to ESP in order to allow more efficient solutions to services and communities discovery and flow management tasks in general, like the "return" behaviour and protected tags, explained later.

4.1 WASP packets

WASP uses the *active packets* paradigm: each packet contains its own code and the data on which it can operate. WASP code and data can be stored in the payload of an IP packet or it can be piggybacked on another packet as an IP header extension. The packet's code consists of up to 256 *microbytes* for a WASP *Virtual Processing Unit* that will eventually lead to a packet control instruction telling whether the packet should continue towards its destination, return to its source or be dropped. The data part of the WASP packet is available as a 128 byte of RAM to the VPU during packet interpretation. Other parts of the packet (code, IP header, payload) are not alterable by the WASP code and only the IP header is readable. Figure 2 shows how node and packet storage areas are viewed by WASP code.

4.2 The WASP node

A WASP node has several *Ephemeral State Stores* that associate 64 bit keys with small, fixed-size data into *tags*. Each ESS is bound to a *Virtual Processing Unit* that processes the WASP packets on a given *location* like e.g. "incoming on eth0". The VPU state is reset everytime a new packet is processed, which means that all the communications and exchanges between packets will occur *in the ESS* associated with the VPU.

Each VPU on the node exports a few information for WASP packets, like its IP address, netmask, the local node time, etc. Outgoing VPU will also export interface-related information like queue length and number of packets sent. These environment variables appear as a bank of read-only memory for the WASP VPU and allow the WASP programmer to design various monitoring or self-adapting services.

4.3 Super Packets and Protected Tags

In existing applications using ESP router, there's no need for access control to tags. It is simply assumed that the each source picks up a random 64 bit word and uses it

Table 1. Relative timings for ESP operations processing in CPU cycles, sorted by ESS accesses

operation	ESS	WASP, nocache	WASP, cached	WASP, MAPping	native ESP
forward	0	129	123	-	-
compare	2	549	543	-	316
count	2	721	592	586	349
collect	4	1245	958	842	633
rchild	6	2058	1845	1509	775
rcollect	8	2980	2394	2020	1091

as a key. Chances that two sources randomly pick the same (publicly available) tag and send packets over routes that cross the same router (otherwise no collision occurs) are virtually nul. When using WASP to advertise services, however, participants are required to use a *well-known* tag value that both service advertiser and service user will put in WASP packets. Unfortunately, we cannot safely use public well-known tags as an attacker could hijack the traffic of a given operator to its own network by advertising his machine as a proxy on the operator's network.

To solve such problems, WASP introduces *protected tags* that can only be modified by *super packets*. The node tells whether a tag is protected or not by checking its key against a specific pattern, and will allow writes to such tags only to packets that are marked 'trusted' in their WASP header. All a network manager will have to do in this case is (1) filter out WASP super packets at ingress nodes from the outside and (2) use super packets to advertise services within his own network.

5 Performance Evaluation

These tests are based on the linux module version of ESP software, running on a 1GHz Pentium 3 machine with default compilation options. Timings were measured using the internal *time stamp counter* of the processor, averaging on 1000 tests to avoid any unwanted side effects of caches, etc. The virtual node state is maintained such that the same (longest) code sequence is evaluated at each iteration. For each of the five ESP instructions on that distribution (*compare*, *count*, *collect*, *rchild* and *rcollect*), we wrote an equivalent *WASP* packet. Note that in order to achieve good performance, it is usually required to tune packet code so that the data organization better suits the instruction flow, as illustrated by Table 5.

One key feature for fast interpretation of WASP code will be how good the interpreter is at avoiding repetitive access to the ESS, and one way to achieve this is by *caching* intermediate results. Tests carried were in favour of very small caches (1 entry) since more complex policies tend to eat all the cache benefit in their initialization.

Alternatively, we can offer larger values for each key in the ESS. Instead of having a single 64-bit word, we now allow a whole memory bank of 32 bytes which can be *mapped* in the VPU's memory. In complex operations like *rcollect*/*rchild*, the state we process is no longer atomic, but instead consists of tuples. While ESP then requires one key per field (and thus one ESS lookup at least per field), WASP allows a few fields to be grouped together as long as they fit one 32 byte bank.

Our tests with the Pentium-based implementation shows an improvement of 18% (two variables per bank) to 35% (four variables per bank) in the processing time as soon as several variables need to be updated, and we expect improvement to be even more important on saturated ESS storage (e.g. when collisions occur inside of the ESS hash table).

6 Conclusion and Future Work

We have presented a lightweight active platform that combines advantages of ESP's per-node storage and SNAP's safe and efficient language. Despite its use of a bytecode interpreter instead of native code, our work still shows execution performance of only 150% to 200% of corresponding native code and is much more generic than the existing ESP framework.

The proposed platform elegantly solves the problem of locating available third-party service providers. We also expect that it could also help peers of a community to find each other, even without the help of a "pong server" and we are investigating the possibility of having *private* tags that would be restricted to packets from the same 'protocol'.

Acknowledgment

We would like to address special thanks to Jiangbo Li from Kenneth L. Calvert's team for having so kindly replied to all our questions related to ESP.

This work has been partially supported by the Belgian Science Policy in the framework of the IAP program (Motion PS/11 project) and by the E-Next European Network of Excellence.

References

1. D. Wetherall, A. Whitaker: *ANTS - an Active Node Transfer System. version 2.0* http://www.cs.washington.edu/research/networking/ants/
2. J. Moore and S. Nettles: *Towards Practical Programmable Packets*, In Proc. of the 20th IEEE INFOCOM. Anchorage, Alaska, April 2001.
3. E. Nygren, S. Garland, and M. Kaashoek: *PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems*, In Proc. of IEEE OPENARCH, pp. 78-89, New York, March 1999.
4. K. Calvert, J. Griffioen and S. Wen: *Lightweight Network Support for Scalable End-to-End Services*, in Proc. of ACM SIGCOMM, pp. 265-278 Pittsburg, PA. August 2002.
5. Jonathan T. Moore: *Safe and Efficient Active Packets*, Technical Report MS-CIS-99-24, University of Pennsylvania, October 1999.
6. S. Martin and G. Leduc: *A Dynamic Neighbourhood Discovery Protocol for Active Overlay Networks*, in Proc. of IWAN, pp. 151-162, Kyoto, Japan, December 2003.
7. R. Braynard, D. Kostić et al. *Opus: an Overlay Peer Utility Service*, in Proc. of the 5th IEEE OPENARCH, pp 168-178, New York, June 2002.
8. J. Touch and S. Hotz, *Dynamic Internet Overlay Deployment and Management Using the X-Bone* in Proc. of ICNP 2000, Osaka Japan, pp. 59-68.
9. Intel Corporation *The IXP1200 Hardware Reference Manual*, August 2001.