Interpreted Active Packets for Ephemeral State Processing Routers

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. We propose WASP (*lightweight and World-friendly Active packets for ephemeral State Processing*), a new active platform based on *Ephemeral State* designed to allow bytecode interpretation on programmable datapath elements. We designed WASP to be a good compromise between flexibility (e.g. offering solutions in quality-adaptive multimedia flows, service discovery or mobility support) and safety (i.e. protection of router and network resource).

1 Introduction and Motivations

With the emergence of network processors, the way we design active networks has evolved [14]. Older active platforms like ANTS [1] offer fully-featured environment supporting complex functions like transcoding video flows, redistributing a packet towards a collection of targets, etc. In constrast, SNAP (*Safe and Nimble Active Packets* from the SwitchWare project[5]) and more recently ESP (*Ephemeral State Processing* router designed at University of Kentucky [4]) stress that active processing should remain safe and efficient in addition of being flexible (in a word, *practical* [2]).

WASP is an attempt to merge the benefits of these two platforms to offer a better compromise between users and network operators' expectations about active networking. The WASP router keeps the focus on *routing* the packets, with the option of performing very simple tasks (e.g. that are "too cheap to measure" compared to packet forwarding) that can help applications in end-system to take better decisions or locally improve flow efficiency.

In other words, we can express the constraint we put on WASP design as follows:

User-Friendly: WASP should offer significant programmability while allowing the end-user to know to what extent he can trust what he gets from the active network.

Network-Friendly: WASP should not become a nuisance to networks (and operators). It should require no configuration from the operator and the network load it produces should be predictable.

Router-Friendly: Active packets should not be able to harm a router nor degrade its performance.

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

1.1 The Need For Speed

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. With the appearance of network processor devices that offer datapath programmability at rates of up to 1Gbps, we may expect a network environment where programmable nodes are not simply end-systems in an overlay but even border routers in a transit domain.

A simple *active packet* crossing an active node will incur different types of time-consuming operations, among which *delivery* to the software component that will evaluate it and *unmarshalling* of high-level language abstractions can be a significant share (up to 42% and 32% of the smallest ANTS capsule respectively, according to [10]) of the total processing time. Considering these results, second-generation active platforms have migrated from the user-level space to kernel level and tend to operate directly on packet data rather than requiring serialization/deserialization between the representation active code uses and the one stored in the packet.

Because it is extremely small, the WASP microbytes interpreter can be made safe enough to run with the same privileges as the "fast path" of the router. Moreover, it doesn't require any marshalling cost as it directly operate on the packet as if it were flat memory rather than trying to assign strong types to data objects.

1.2 How Network Processors differs from Generic-purpose Processors

We developed the WASP platform with two target platforms in mind: regular PC systems (where it could be for instance running as a Linux kernel module) and routers equipped with *network processors*, where it is possible to implement efficiently solutions that require custom operations on packets. Network processors typically include a general-purpose processor (for control plane purpose), specialized co-processors (for hashing or trie lookup) and dedicated RISC cores for programmable datapath on a single chip. These chips try to avoid SDRAM lookup latencies[1] by providing closer, faster (and smaller) storage for both data and code and often access multiple words in a row. These hardware considerations have an impact on the design we can implement efficiently on such system.

In addition, in the Intel IXP family ([9]), each execution unit (called a *microengine*) has very small instruction memory (2K-4K) whose content cannot be reprogrammed while the microengine is in use. It is thus quite unpractical to implement a platform that would cache native code at the *microengine* level. Moreover, if one wishes to implement a bytecode interpreter, it will have to be simple enough to fit a few microengines.

1.3 Third-Party Services: an Hybrid Networking Approach

Literature on active applications have shown how end-users could benefit from high-level operations such as hierarchical HTTP caches and multimedia flow transcoding depending on terminal/network abilities. A network operator who is willing to offer such "value-added services" to customers will however face the problem of client-side

[1] According to [6], it will take up to 750 cycles of the IXP1200 microengine

configuration and service advertisements. In our previous work [8], we have shown how the WASP platform could be used to help deploy such services through a fully decentralized, configurationless discovery scheme.

The key idea of that service location facility is that, if sufficient WASP nodes are deployed in a network domain, nodes that are willing to offer a given service S simply need to leave information in WASP routers so that, when customers establish a new connection, they can automatically detect whether a given service is available or not and which "proxy" node should be contacted to get the service. In terms of speed improvement, this hybrid approach allows the operator to draw the full power of nodes that hosts services with limited overhead on routers, taking benefits of both *component-based* active networks for the services implementation and *active packets* for services discovery.

2 A Router-Friendly Platform

Even if WASP is based on active packets, it is much more restricted than general-purpose *capsules*, so that user's bytecode cannot waste router's resources. For instance, WASP bytecode language prohibits backward jumps and all instructions have predictable execution time, which makes packet processing time trivial to control (as shown in SNAP [5]). If such a restriction is impractical for general-purpose services, it perfectly fits the lightweight control tasks that WASP will have to perform. While alternative solutions exist, such as associating a counter to any backward jump in the code, we believe the benefit we could get in WASP is not worth the additional management required.

2.1 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. For instance, we could drop *B frames* of an MPEG video stream if the *I frame* they refer to has been dropped by the node or if it is likely to be dropped, for instance due to a congested output link. This requires *I frame* to leave information on the router status for further frames. 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. It has been shown in ESP [4] that memory will be much easier to manage in the *ephemeral state* approach, that is if the store only keeps data for a constant period (10 seconds), *regardless of how frequent the data is referenced during that period*.

If we also ensure that 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 checking if the router will have sufficient resources to process an additional flow simply requires that the router checks how many different

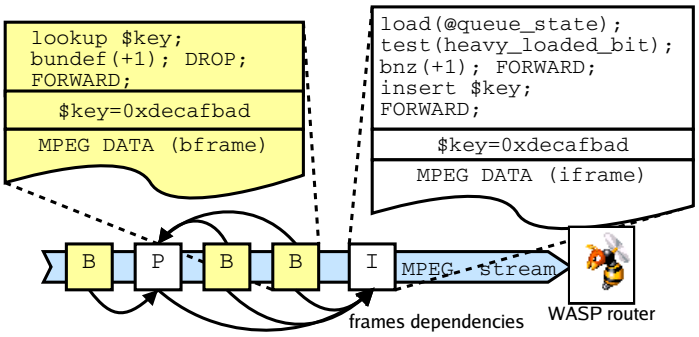

Fig. 1. *WASP code attached to MPEG* I frames *and* B frames

slots are used by the flow. Those small, fixed-size data slots that the node associate with a key for a fixed amount of time are called *tags* in the ESP terminology. Note that no access control is required for tags. It is simply assumed that each source picks up a random 64-bit value and uses it as a key. The chances that two sources randomly pick the same tag and send packets over routes that cross the same router in a 10 seconds timeslice (otherwise no collision occurs) are extremely small.

2.2 A First Example

The above decisions may sound excessively restrictive, but the platform we build on it can still be sufficiently expressive. Figure 1 illustrate a WASP implementation of the selective frame filter [12]. The code for *I frames* checks a bit of the interface state to evaluate the load on the outgoing interface (set by a RED queue manager, for instance) and, depending on the result, leaves (`insert`) a tag demanding *B frames* to be dropped.

All the *B frames* depending on a given *I frame* will carry the same *unique identifier* (e.g. `0xdecafbad`) generated by the source and use it to check (`lookup`) the presence of the tag in the router's store. Note that this scheme could easily be extended so that, for instance, a *B frame* that faces a 'drop request' propagates this information backwards to the source. We could also use an *I frame* code that would compare the local time with the reception time of the last *I frame* to see if the deadline is still achievable.

3 A Network-Friendly Framework

Even if we take care of router resources, an ill-intentioned active packet could easily create an avalanche of clones to overload its destination. Among 'first generation' active platforms, PLAN [11] was the only project that addressed this issue by making sure all children's resource counters receive a portion of the parent's counter. Unfortunately, knowing what initial bound should be allowed remains a complex issue.

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, but they can *drop* themselves or *return* to their source. If we focus on applications like service discovery or server load balancing, we have no real need for more: we can store the new destination in the active

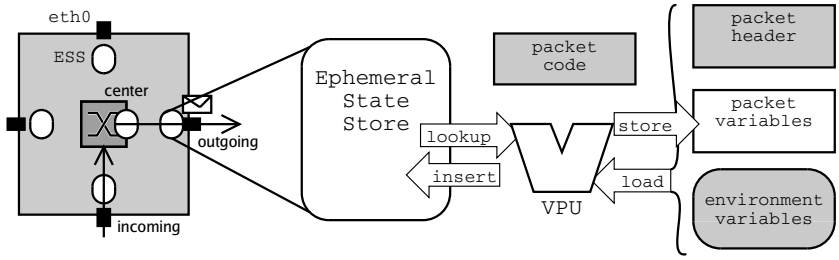

Fig. 2. *A WASP router and WASP execution environment. Gray items mean the VPU has read-only access to the resource*

packet, *return* the packet towards its source and let the source issue a new connection attempt to the real destination.

In the case of content distribution networks, however, being able to change the destination address could have a great impact on performance, especially if we are allowed to swap a *multicast* address to unicast addresses. Using this scheme, it becomes possible to take advantage of multicast routing *where it is deployed* rather than requiring a consistent deployment along the whole path.

From the network operator point of view, such rerouting needs to be carefully handled so that it does not lead to packets looping endlessly within a domain or "ping-ponging" between two domains. For instance, by the action of rerouting, a WASP packet should never require an additional IP table lookup and it may never lead to sending a packet back on the interface it is coming from. The problem of rerouting is further discussed in section 6.

4 The WASP Platform

WASP is derived from ESP router[4], which consists of the router logic and *Ephemeral State Stores* (ESS) containing *tags* that packets access based on 64-bit *keys*. As shown on Fig. 2, ESS are either bound to a network interface or to the "center" location, and active packets that cross the router can only request interpretation at 3 logical locations: incoming interface, center and/or outgoing interface.

Each ESP packet requests the execution of one of the pre-defined operations on certain tags. Unfortunately, those operations remain tightly bound to a specific application domain (multicast) and non-trivial protocols require almost dedicated operations. 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 [5]. We also extended the packet control operations, access control semantics on the tags and access to node-specific state.

4.1 WASP packets

WASP uses the *active packets* paradigm: each packet contains its own code and the data on which it can operate. The evaluation of packet code (up to 256 bytecoded

micro-instructions called *microbytes*) terminates when a *packet control microbyte* is encountered, which tells the router what to do with the packet. In addition to "forward" and "drop" semantics, WASP allows the packet to be sent back to the source at any router, which can be useful when a quick feedback of a discovered state is required (e.g. filtering more packets in the router ahead of a congestion point).

During interpretation, the data part of the WASP packet is available as a 128 byte region of random-access memory and only the IP header is readable. Other parts (WASP bytecode, other IP options, transport payload) are unavailable to WASP code.

4.2 The WASP node

Each WASP node has a certain number of *Ephemeral State Stores* that will associate 64-bit keys with small, fixed-size data into *tags*. Each ESS is associated with a *Virtual Processing Unit* (VPU) that processes the WASP packets. Since all exchanges between packets occur in the ESS, there is no need to store VPU state between the evaluation of two packets. This greatly simplifies the synchronization problems, even on a multi-processor system, since it means we can bind VPU data to one real CPU rather than to an ESS.

Before a VPU starts evaluating a packet, it retrieves the node and interface *environment variables* and exports them as banks of read-only memory to WASP code. Those variable will typically contain the node IP address, netmask, local time, etc. plus statistics about the current interface (recent packet transmission statistics, queues status, etc.), which can be useful for applications sensible to network conditions.

Considering the restricted resources of network processors, we tried to keep the design of WASP's virtual processor as simple as possible, which makes it look more like an embedded microcontroller than a modern microprocessor, from an architectural point of view.

- Work registers are organized following the 'accumulator and stack' approach, leading to smaller encoding and better use of hardware registers.
- Once the *index* register has been loaded, any memory reference can either stay on that index or advance to the next word. With an appropriate ordering of data according to code sequence, this may save up to 50% of code size for the ESP instructions involved in data aggregation service [15].
- Only simple ALU and shifting operations are allowed. The VPU has, for instance, no support for floating point values, multiplications/divisions, or signed arithmetic.

4.3 More Efficient Access to ESS

Among all microbytes, interactions with the ephemeral state store will be the most important to tune, as they will likely be the most costly operations the VPU will have to handle. It is clear that we'd like to avoid repetitive hashing in a lookup-then-update cycle, for instance. In native implementations, once the hash table has been looked up, resulting memory references are kept for further updates. In WASP VPU, those "resolved pointers" are stored in a cache transparent to the bytecode programmer. We

	no caching	full	small	mapping	native
count	721	637	592	586	349
collect	1245	1082	958	842	633
rchild	2058	1830	1845	1509	775
rcollect	2980	2438	2394	2020	1091

Table 1. *Comparing caching policies. Timings in CPU cycles on 1GHz Pentium III*

tested two cache policies, *small caching*, where only the last resolved pointer is kept, and *full caching*, which keeps every resolved pointers.

Moreover, previous work with *ESS*-based programmable nodes has shown that non-trivial operations (including *rchild* and *rcollect* used for the robust aggregation service) may require 3 or 4 logical variables in the ESS, leading to increased processing cost due to repeated search in the ESS and repeated access to SDRAM.

WASP solves these issue by allowing larger values (namely 32 bytes) to be stored in the ESS, and through a `map` microbytes that makes a whole ESS value appear as a memory bank for the VPU. The packet can then access individual bytes/words of that bank with no extra key resolution until another `map` forces the bank to be written back in the ESS. Even on the PC implementation, mapping larger memory banks has allowed us to reduce execution time by about 30%, and we expect even higher improvements on IXP architecture thanks to burst transfers with SDRAM. It is also interesting to note that, due to overhead in the ESS entries, allowing 8 times more storage leads to better memory consumption as soon as we have an average of at least 2 related values.

4.4 Preliminary Performance Results

To validate our assumptions, we compared the execution time of ESP operations on a Pentium III processor, interpreted by WASP using different access policies, against ESP's native implementation (see Table 1). Note that the *small caching* policy behaves better than *full caching* here. This can be explained by the fact most of ESP operations (as described in [15]) don't look up for a given variable more than once and that updates can be done before another lookup is issued. The cost for setting up and maintaining a more complex policy such as *full caching* is thus greater than the benefit one can expect from the cache hit ratio. The *mapping* scheme is clearly giving even better results once the bytecode has been rewritten to use the `map` instruction, approaching an execution time of 200 to 150% of the *native* implementation provided by University of Kentucky.

Note that even if *interpretation* with WASP is twice longer than execution of native code, this represents only a small fraction of the code that is actually executed to process a packet. For instance, on a Linux router running at 300MHz featuring the ESP/WASP module (with small caching policy), the `ESP:count` packet took an average $69.8\mu s$[2] while processing `WASP:count` took $77.6\mu s$ – only a penalty of 10%,

[2] those timings are obtained using timestamps returned by *libpcap* on a machine running Linux kernel 2.4.18, which forwards a ping with an average latency of $48\mu s$ and takes $99.8\mu s$ to reply to a ping.

compared to 69% suggested by Table 1. Comparatively, the same packets take respectively $23.6\mu s$ and $24.3\mu s$ to be *just forwarded*. Moreover, a more complex packet like `WASP:collect` took $80.0\mu s$ for processing and $25.0\mu s$ for plain forwarding. Finally, a WASP packet of the same size as `WASP:count` that simply contains the `forward` microbyte took $66.6\mu s$, which means most of the overhead is in packet checking, initializing and finalizing rather than in instructions processing.

5 Trustworthy Storage for the End-User

As soon as WASP is used to locate services, packets need to use a *well-known* key to access information other participants might have left in routers. Such a well-known key can be for instance produced by hashing a service name, which makes them easier to guess from an external attacker than random keys of section 2.1. Therefore WASP introduces *protected tags* that can only be modified by *super packets*. If the domain operator ensures that no super packets can come from outside, the end user can be sure that the information bound to the tag has been set up by the domain operator. The node determines whether a tag is protected or not by checking its key against a specific pattern[3], and will allow writes to such tags only to packets that are marked 'super' 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.

5.1 Hash-Requesting Packets and Private Tags

When participants and attackers can come from the same domain, protected tags are no longer helpful. For such cases, WASP offers *private* tags, which works like protocol-private data in ANTS. Unlike other tags, the application programmer has no direct control on the key that will be used for private tags. Instead, the WASP node will hash the code contained in the packet and use the result as the *private key* for that packet, which is kept secret by the router. To make sure that regular packets do not attempt to use brute-force scan, private tags have an identifiable prefix and any attempt to use keys with that prefix explicitly will abort packet execution.

If the hash method is carefully chosen (e.g. a one-way hash like MD5), it means that packets will have access to the *same* private space *only* if they have the same code, which means we are sure they play the same game with same rules. Under those circumstances, an attacker can only hope to break the protocol by sending more (or less) packets than expected by the protocol – which a properly designed protocol should handle anyway. Note that the implementer of a WASP node is free to use any hash method that best suits its hardware as the resulting hashes are used only on the computing node. The only rule is that packets with the same hashed part operate on the same private tag and that packets with different hashed parts operate on different private tags.

[3] highest 8 bits are all 1 in current implementation

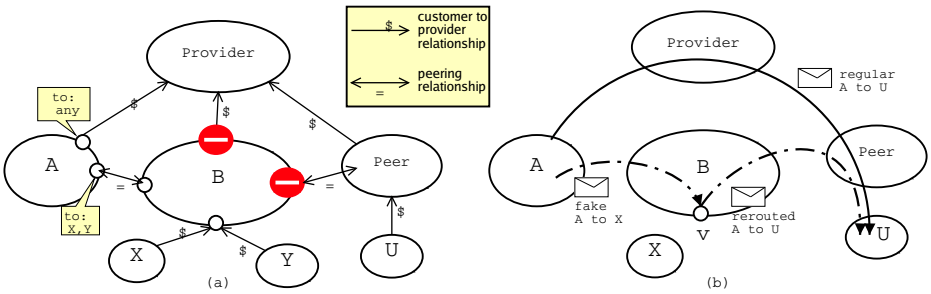

Fig. 3. *Typical interdomain policies for domain A (a), broken by blind rerouting (b)*

Note that routers that cannot afford MD5 could be allowed to use cheaper algorithms (e.g. CRC) by mixing the bytecode with a locally-generated random number and still be able to safely assume that protocol's privacy cannot be broken.

5.2 World-readable, Protocol-writable tags

While private tags guarantee that a collection of participants will modify state in the router following a common set of rules (e.g. the protocol), their cost may not be acceptable for packets that just need to follow the decision without altering the state (e.g. a multimedia stream). Each packet would also have to carry the *whole* protocol so that it receives the same hash value, regardless of what part of the code is useful for itself.

As a result, the `expose` opcode allows a hash-requesting packet to have its private state accessible read-only as a protected tag. The result is a new ESS tag that contains a *link* to the private tag, which is transparently resolved by the VPU when a packet tries to read it. Writing to an exposed tag via a link is of course not allowed.

Note that the presence of a link only tells that it exposes private data, but not *what protocol* exposes them. It will thus be up to the protocol designer to ensure that the key used for exposing the data cannot be guessed by an attacker before the link is created. A simple way to achieve this is to generate the key from a random number on the router and inform participants of its value *after* data has been exposed.

6 Rerouting Packets

While considerably augmenting the flexibility of the WASP platform, packet rerouting raises a number of issues for both network operator and end-users. The main concern for the end-user will be to ensure that packets still reach the expected destination, even when re-routing applies. When the sequence of destination addresses to the final target is given explicitly in the packet's variables, the offered service has the same security semantic as loose source routing in IP. When that sequence is retrieved from ESS, however, we need to ensure that no one is trying to abuse the end-systems to gain an intermediate position on a specific data flow. We are confident that protected/private tags should however help the end-user build reliable rerouting-based applications.

From the operator's point of view, the main difficulty comes from the fact that generally speaking, we do not want to allow any packet to be received from any link. Business agreements with other peer domains, for instance, may only allow a domain A to use link to domain B to reach B's clients, but not to reach other peer domains or providers of B (see Fig. 3a). Nowadays, most of these agreements are enforced by filtering routes advertised by BGP rather than by filtering packets, but blindly enabling rerouting of WASP packets could lead to situations where a packet leaves A with a destination address falling in one of B's clients and then reroute itself to another *peer* domain on B's ingress router, thus cheating the business model (see Fig. 3b).

We propose to solve those problems by means of *invitations* left in the ESS by former packets. When a WASP packet executes on an interface VPU, it can create a new tag carrying its source address by means of the `invite` opcode. The binary pattern of the key used with *invite* tells the VPU that the value can be safely used as a redirection target. Depending on how the interface is configured, the `reroute` opcode will accept either any target value (e.g. for customers-ingress links) or will be restricted to protected tags and invitations (e.g. for any other link). This way, "ping-ponging" between domains is no longer possible if all egress interfaces restrict rerouting to invited destinations. An invitation to address Y present on the interface means that the peer router for that interface has once sent a packet coming from Y, and thus it should be able to route another packet towards Y properly.

We can also prevent cheating on the business model if non-client (guest) packets can leave invitations only on *incoming* VPU of ingress routers. More precisely, if we make sure that WASP packets from peers and providers are tagged as guest when they reach the *center* (see Fig. 2) of their ingress router and that *invite* opcode is not allowed for guest packets, then a packet p received on a non-client ingress interface that is targetted to a client domain can only be delivered to a client domain. By contradiction, suppose V is the first VPU where rerouting changes packet p's destination towards a non-client destination U. This is only possible if an invitation to U is present in V, however:

- if V is on a core router, it cannot have the invitation since guest packets can only leave invitations on their ingress VPU (unless source addresses are spoofed)
- if V is on a border router, it implies V is bound to an outgoing interface, say *itf*, towards domain U. Note that p can only reach that hypothetical interface if *itf* connects to both clients and non-clients domains (likely to be a configuration error).

In Figure 3, note that B is not protected if A allows blind rerouting on its egress interface to B (we could easily disable blind rerouting on output interface, anyway). If both A and B configure rerouting properly, malicious clients from A cannot lead A to a situation where it unwillingly misroutes packets through B.

Note that even if rerouting does not, by itself, allow source spoofing (which is the root of most DDoS attacks [13]), it might disturb tools based on header-hashing for traceback of packets used to react to those attacks. Allowing interoperations between rerouting and traceback might include storing previous destination in a field of the WASP packet or keeping traces of applied rerouting on routers and will be an interesting challenge for future work.

7 Conclusion and Future Work

We proposed a new platform that combines safety of ephemeral state processing and flexibility of a bytecode language. While involving strong restrictions on programs one can write, WASP can still address a large range of problems and it can be used as a "helper" tool for more complex solutions that require a more "applicative layer" approach. Still, for the network operator, WASP is safe and offers strong guarantees on processing time, memory requirements and network link consumption.

For the end-user, WASP gives enough programmability for most per-packet control operations. Moreover, WASP is clear concerning what may happen and what may not: the active network will never, for instance, alter source addresses or payloads, nor will it reroute packets if not explicitly requested. It would be interesting to investigate further how the *receiver* can be given more control on what WASP function can/must be supported by received flows, and whether it could help hosts protect themselves against DDoS, spam, etc.

The preliminary benchmarks and the design choices of this platform give us good hope that an implementation on network processors will be able to sustain high throughput of active packets, even if a real implementation on IXP network processor is still required to measure under which proportion of active packets wire speed can be achieved.

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.

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. K. Calvert, J. Griffioen, N. Imam and J. Li: *Challenges in implementing an ESP service*, in Proc. of IWAN, pp. 3-19, Kyoto, Japan, December 2003.
7. 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.
8. S. Martin and G. Leduc: *An Active Platform as Middleware for Services and Communities Discovery* in Proc. of 2nd Int. Workshop on Active and Programmable Grids Architectures and Components, May 2005, Atlanta, USA, LNCS, 3516, pp. 237-245, Springer-Verlag
9. Intel Corporation *The IXP2400 Hardware Reference Manual*, November 2003.

10. Wetherall, D. *Active network vision and reality: lessons from a capsule-based system*, Operating Systems Review, vol.33, ACM, Dec. 1999. p.64-79.
11. M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles: *PLAN: A programming language for active networks*. In Proc. of ACM ICFP'98, pp. 86-93, Sept. 1998.
12. S. Bhattacharjee, K. Calvert E. Zegura, Technical Report GIT-CC-96/02, *On Active Networking and Congestion*, Georgia Institute of Technology. ftp://ftp.cc.gatech.edu/pub/coc/-tech_reports/1996/GIT-CC-96-02.ps.Z
13. T. Dübendorfer, M. Bossardt and B. Plattner: *Adaptive Distributed Traffic Control Service for DDoS Attack Mitigation*. In Proc. of SSN 2005, April 2005, Denver, USA.
14. James P.G. Sterbenz, *Intelligence in Future Broadband Networks: Challenges and Opportunities in High-Speed Active Networking*. In Proc. of IEEE IZS 2002, Zürich, Feb. 2002, pp. 2-1 – 2-7
15. K. Calvert et al. *ESP Packet & ESP Instruction Specification*, Technical Report, http://protocols.netlab.uky.edu/˜esp/documents/esp_spec.pdf, University of Kentucky.