NDIV: a low overhead network traffic diverter

2014/09/26

Willy Tarreau <willy@haproxy.com>

HAProxy Technologies
ALOHA R&D
http://www.haproxy.com/

Initial idea

Requirements appeared around 2000 :

- Use web-like traffic to test firewalls, proxies, load balancers, anti-viruses
- Few available products, and with extremely poor performance

⇒ **Let's build the missing parts** !

First attempt :

- **Oct 2000** : birth of the "*inject*" client, for use with *thttpd* or *Apache*

```
bash-4.2$ /data/git/public/inject/injectl4 -u 10 -G 127.0.0.1:8000/
   hits ^hits hits/s  ^h/s     bytes  kB/s  last  errs  tout htime  sdht ptime
   8256  8256   8231  8231   1205376  1201  1201     0     0 7.2 0.7 7.2
  16336  8080   8163  8096   2385056  1191  1182     0     0 7.1 0.5 7.1
  24536  8200   8170  8183   3582256  1192  1194     0     0 7.0 0.2 7.0
^C
```

- **Nov 2000** : birth of the "*sizesrv*" server by *Benoit Dolez*, for use with *inject*

```
GET /3k&t=100ms HTTP/1.0
HTTP/1.0 200 OK
Content-length: 3072
```

- **2003** : Netfilter benchmark : *inject* & *sizesrv* do not scale enough.

⇒ **If only I could run a single dumb server and have all machines for *inject* !**

First Observation

Shortcomings :

- Problem is to support **very short** requests at high rate
- **Connection** processing overhead is the first cause
- **Packet** processing overhead is another one

⇒ **Can we run stateless and avoid the connection overhead ?**

Acceptable tradeoffs :

- Only support **short** connections (simpler state machine)
- No security considerations (we're benchmarking)
- Undefined behaviour for non-HTTP traffic is acceptable
- But **must comply** with TCP specs (must work with any client)

First attempt at a design

Where to store the state

- TCP flags ? ⇒ OK
- TCP sequence numbers ? ⇒ OK
- TCP timestamps ⇒ would be nice but not acceptable

First attempt at a design failed, so back to continuing improving my tools :-(

- **Sept 2004** : "*inject*" becomes multi-process
- **May 2006** : "*httpterm*" replaces and improves sizesrv

⇒ Still missing support for line rate on very small objects

New ideas and hopes in 2013 at home

- Working at improving network performance on Armada370/XP
- Noticed that Marvell's Neta Ethernet controller is **line-rate** capable
- CPU is not *that* fast and full TCP stack + *httpterm* are much slower
- Plat'home offers me an **Armada XP**-based **OpenBlocks AX3/4** :

⇒ **Already dreaming about line-rate sniffing :-)**
... But failed to implement something working outside of /dev/shm :-(
⇒ **Starting to imagine a simple framework to call external tasks : NDIV**

New ideas and hopes in 2013 at work

Testing other packet processing frameworks :

- netmap - http://info.iet.unipi.it/~luigi/netmap/
- PF_RING™ - http://www.ntop.org/products/pf_ring/
- Intel® DPDK - http://dpdk.org/

Observations :

- all of them are pretty fast on packet processing / forwarding
- all of them are designed for fast **data planes in userland**
- `netif_rx()` is not an option for fast delivery to **local stack**

⇒ **Need for a completely different design for fast local delivery**

Requirements for a new packet processing framework

Minimum requirements for the framework :

- **inspect** received packets with low overhead
- **pass** received packets to local stack with almost no overhead
- **drop** received packets at almost not cost
- **respond** with a crafted packet as fast as possible
- come with a very simple example application to test it

⇒ Same requirements as for the stateless HTTP server, let's try again first !

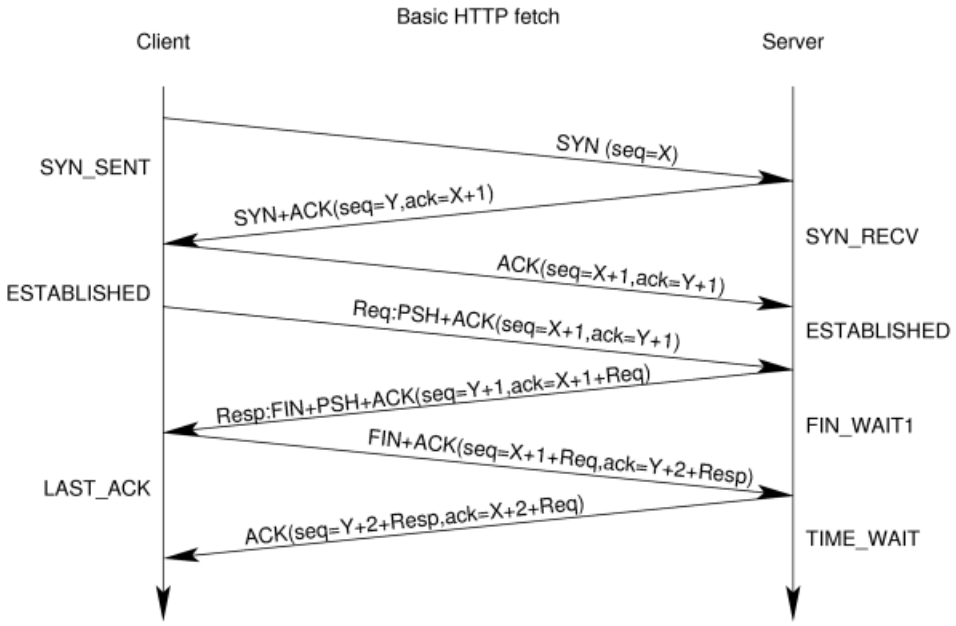
Revisiting the stateless HTTP server

A very quick reminder about TCP's basic principles (see RFC793) :

- two opposite, independant, unidirectional streams
- packets carry **flags** to indicate **local status** (SYN, FIN, RST, ...)
- packets have a **sequence number** indicating their **position** in the stream
- sequence numbers reflect transmitted **byte count**
- SYN and FIN flags are seen **only once per direction** and count as **one byte**.
- Initial sequence number is **chosen** by each party when sending the SYN flag
- ACK field indicates **next expected sequence number**
- ACK (almost) any data or SYN/FIN you receive
- **retransmit** anything not ACKed after a timeout

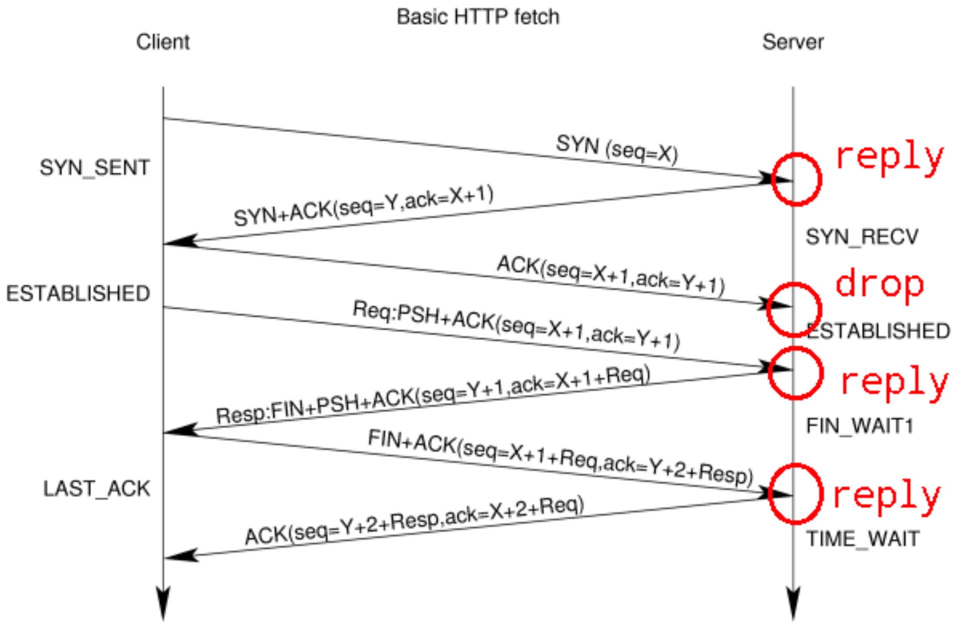
Revisiting the stateless HTTP server

A typical, complete short HTTP connection uses 7 packets



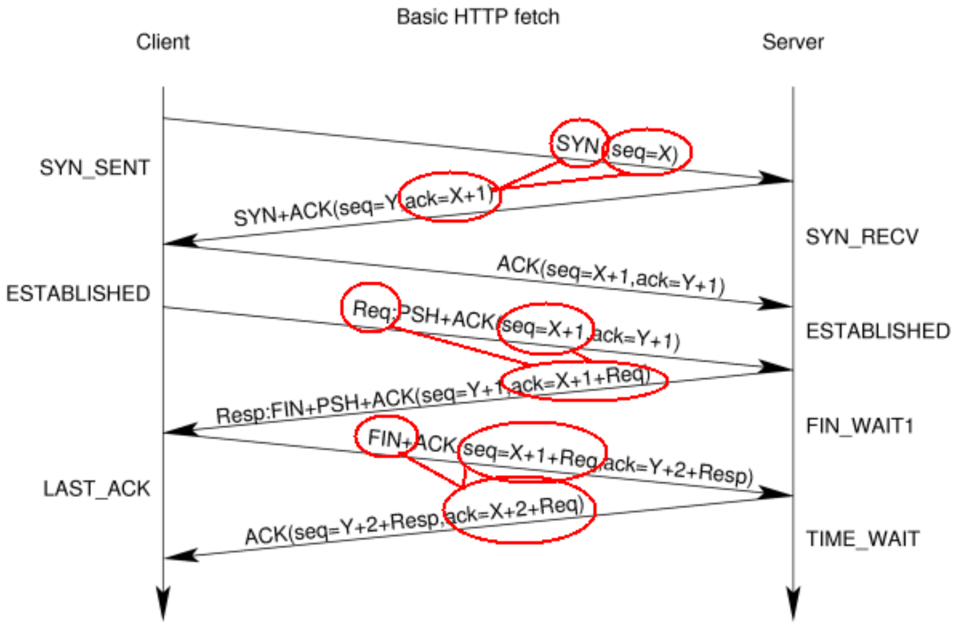
Revisiting the stateless HTTP server

A typical, complete short HTTP connection uses 7 packets



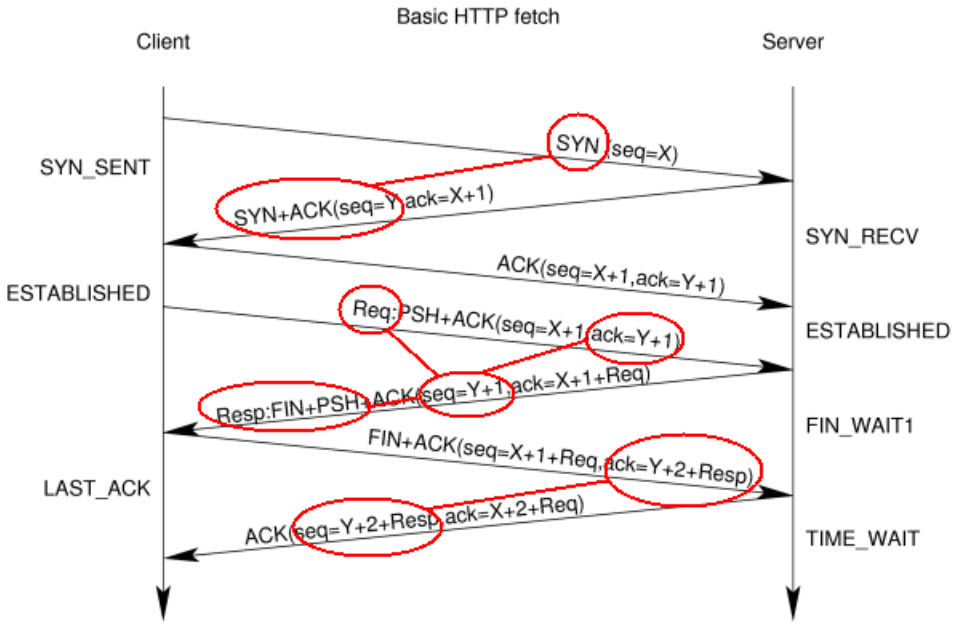
Revisiting the stateless HTTP server

A typical, complete short HTTP connection uses 7 packets



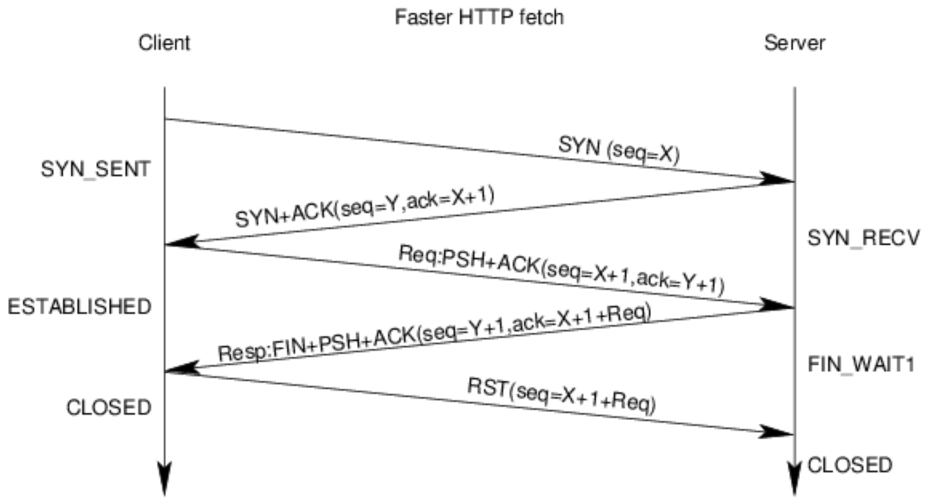
Revisiting the stateless HTTP server

A typical, complete short HTTP connection uses 7 packets



Revisiting the stateless HTTP server

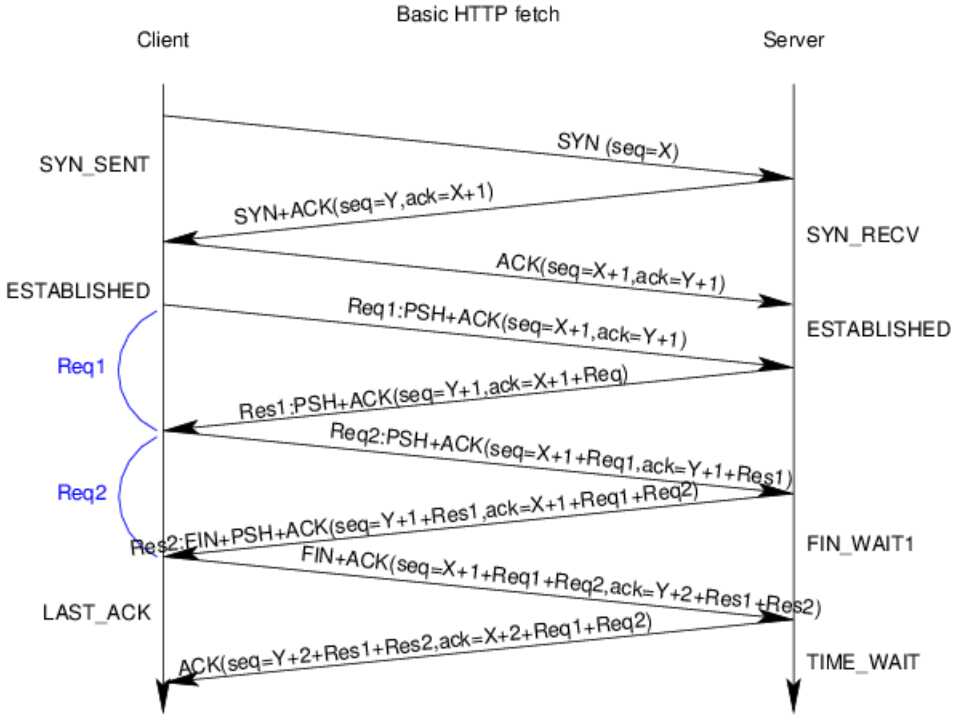
An optimized HTTP connection may be down to 5 packets



⇒ Even the shortest connections are compatible with this principle

Revisiting the stateless HTTP server

With HTTP keep-alive, the request-response loop repeats as long as needed



Revisiting the stateless HTTP server

Conclusion : the design is very simple :

- **drop** anything without data nor SYN nor FIN
- **respond** to SYN with SYN-ACK and a carefully picked sequence number
- arrange **initial sequence number** so that we can recognize them in incoming ACKs
- always send SEQ = last ACK when present
- always send ACK = last SEQ + received_data + FIN + SYN
- send a response when receiving a request
- set the FIN flag on the last response
- **never ACK anything without sending DATA, SYN or FIN**
  ⇒ all lost packets are dealt with by the client
- could even support TCP Fast Open with minor changes !

Revisiting the stateless HTTP server

We need 4 states :

- 0 : **REQ** : waiting for the request, just after the SYN-ACK is sent
- 1 : **ACK** : a client's FIN was received, waiting for ACK of our FIN
- 2 : **CLO** : our data were ACKed but not the FIN yet (rare)
- 3 : **FIN** : our FIN was sent and ACKed

⇒ **These states can be encoded in our sequence numbers echoed by the client**

Note: the original implementation (***slhttpd***) supports 16 states to send large responses, but it was unreliable since a lost client's ACK will not be retransmitted.

Revisiting the stateless HTTP server

We use a few tricks :

- Always send data in multiples of **4 bytes** (leaves 2 bits for state)
- use FIN and `X-Pad` header to adjust the state
- REQ → REQ transition is used for HTTP keep-alive
- ACK must equal REQ + 1 (as our response FIN counts for 1)
- FIN must equal CLO + 1 (same reason)

Revisiting the stateless HTTP server

Two proofs of concepts were made :

- NFQUEUE (userland) : fast and portable development :
  - ~**33.000 conn/s** on the AX3
  - ~**105.000 conn/s** on a Core2 at 3 GHz
- kernel-only dummy interface (Tx path) :
  - ~**42.000 conn/s** on the AX3
  - ~**175.000 conn/s** on the Core2
- Both support `GET`/`HEAD`, send requested response size and parse the `Connection` header
  This is already twice the speed previously achieved using httpterm

⇒ **time to bring the old NDIV hacks back to the whiteboard**

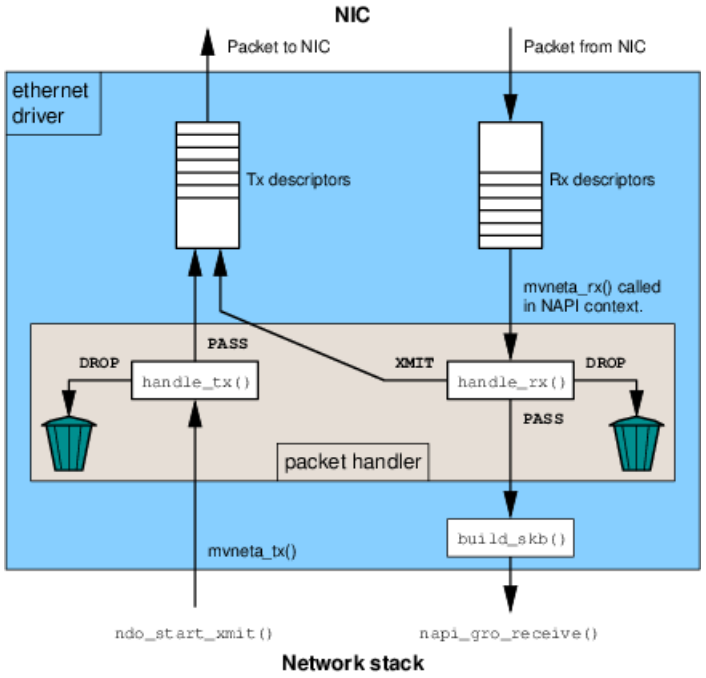
New ideas for the NDIV framework

Based on what the HTTP server and sniffer requirements, and what we saw in other designs :

- CPU **L1** caches are **small**, limit copies to absolute minimum
- Work in **kernel space** to gain direct access to data **without copying** into descriptors
- Most CPUs have branch prediction units, better use **callbacks** than **queues**
- Pass performance-critical information in **registers**
- Modifying network drivers is not *that* hard (learned from netmap)
- Adapt to **best**, not to worst : focus on ideal NICs and make the **driver** fill the gap
- Pass useful L2/L3/L4 offsets to the application to save it from **parsing packets**
- Make it easy for the application to build packets and let the **driver/NIC finish the job**
- Pass **pre-allocated Tx buffers** to the application in case it needs to respond
- Support an `rx_done()` function to **flush pending work**
- Run under **NAPI** to ensure we don't interrupt anyone

Placing the NDIV framework in a driver

Basically one function for the Rx path and another one for the Tx path



Design of the NDIV framework

Everything fits in a single "`.h`" file of ~250 lines (1/3 doc & comments).

```
struct ndiv {
        struct net_device *dev;
        u32 (*handle_rx)(struct ndiv *ndiv, u8 *l3, u32 flags_l3len, u32 vlan_proto, u8 *l2, u8 *out);
        u32 (*handle_tx)(struct ndiv *ndiv, struct sk_buff *skb);
        void (*rx_done)(struct ndiv *ndiv);
};
```

A few hints are used.

- attach: the **pointer** to the `ndiv` struct is stored in `dev->ax25_ptr`
- checking this pointer is enough to know if ndiv is attached
- use a lot of **composite** input/output values to reduce **register pressure**
- make **most commonly** used values **easily accessible** (eg: length on 16 lower bits)
- adjust number of packets really reported to NAPI on Rx

NDIV callbacks

`handle_rx()` is called inside the `poll()` loop under NAPI

`handle_rx()` arguments :

- direct pointers to L2 and L3 (allows holes used to align packets)
- `flags_l3len` : high 16 bits : IPv4/IPv6/other, extensions, TCP/UDP/other, L3/L4 csum validity
- `flags_l3len` : low 16 bits : L3 len
- `vlan_proto` : vlan ID (16 bits) + L3 protocol (16 bits)

NDIV: callbacks

`handle_rx()` output : 32 bits

- Action (2 bits) : SKIP / PASS / DROP
- Output packet length (for XMIT on DROP or changing on PASS)
- IP/TCP/UDP checksumming needed (yes/no)
- L4 offset (for checksum)
- VLAN tag present (yes/no)
- IPv4/IPv6/Other

NDIV: callbacks

`rx_done()`

- Called once after the Rx loop if any Rx done
- Used to wake up user-space, to flush stats or buffers (eg: sniffer)

`handle_tx()` takes an SKB from the local stack

- Only three actions for now : SKIP / PASS / DROP
- Nothing planned to modify the packet yet
- API could be changed to return skb or NULL

NDIV implementation

NDIV was first implemented in mvneta (Armada XP/370)

- Basic work, including very basic XMIT : ~1 day
- Implement checksum/VLAN/IPv6 checks : another day
- DMA API optimizations (implement a single DMA barrier to avoid `dma_unmap()`)
- A per-queue Tx descriptor pool was implemented
- one Tx queue must be locked when doing XMIT.
- reduce lock cost by unlocking after non-Tx packets

⇒ **both OpenBlocks and Mirabox support mirroring 1.488 Mpps (line rate)**

Porting SLHTTP to NDIV

The stateless server was ported to NDIV and run on the AX3

- Simple port from the dummy interface code : ~1 day, ~600 loc
- Only attaches to a destination port range in TCP over IPv4
- No measurable performance impact for communications to/from TCP stack
- Achieves **340.000 connections** per second (test limited by request size) :
  - SYN: 8 + 64 + 12 bytes = 84 bytes
  - REQ: 8 + 14 + 166 + 12 bytes = 200 bytes
  - RST: 8 + 64 + 12 bytes = 84 bytes

340k/s * 368 bytes
= <u>1 Gbps</u>

- and **663.000 requests/s** in keep-alive using *ab*
- Limiting factor is always the request traffic saturating the link!

⇒ **Achives line rate for all sizes on a single CPU core**

NDIV next steps

This design is still **experimental**

- HAPTech ported it to Intel's `ixgbe` driver ⇒ **14.88 Mpps** both ways
- No support for forwarding traffic between two ports
  ⇒ but you can use VLANs and a switch :-)
- Reactive design only - doesn't generate traffic
  ⇒ *pktgen* or `PF_PACKET` hacks still required for generation
- No support for mangling outgoing traffic. Really needed ?
- Not yet implemented on loopback
- Line-rate 10Gbps capture already works - tested!
- Still missing dedicated statistics
- Easier to implement in drivers already using `build_skb()`

Possible applications

Possible applications for **NDIV** include :

- Network testing (eg: with SLHTTPD)
- Measuring latency (via packet rate on a loop)
- Line-rate packet capture, firewalling, pattern matching, ...
- Traffic load balancing (using VLANs)
- Traffic bridging / routing (using VLANs)

Possible applications for **SLHTTPD** include :

- Network equipment validation (proxies, firewalls, routers, load balancers)
- Internet of Things (IoT) ⇒ check sensors over HTTP without a TCP stack
- Serving small static objects (favicon.ico)
- Error pages and redirects

Thanks!

Questions ?

- Complete article here : http://1wt.eu/articles/openblocks-http-server/ (*contains experimental code*)
- Final patches should be available by Q1 2015 at http://haproxy.com/
- For any other question : willy@haproxy.com