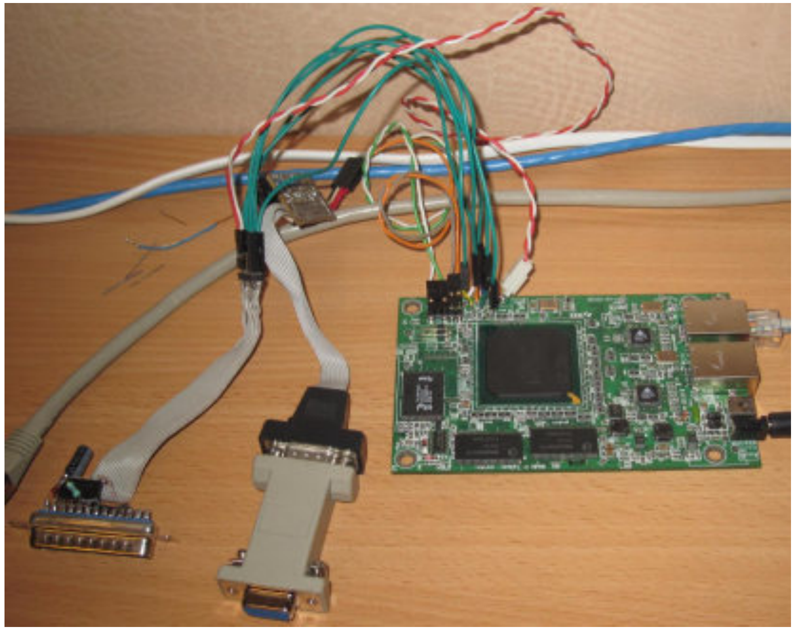
Developping drivers on small machines

2013/09/24

Willy Tarreau
<willy@haproxy.com>
HAProxy / ALOHA R&D
http://www.haproxy.com/

Who am I ?

- Lead architect at HAProxy France
- Long time maintainer of very old kernels (2.4, 2.6.32)
- Always interested in stuffing Linux anywhere :-)

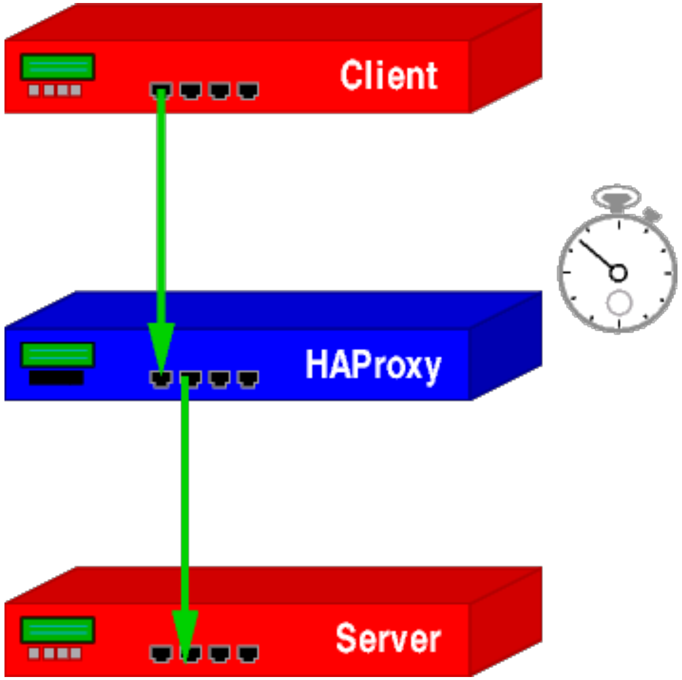


Day to day job - *the good and the ugly*

- Develop, test, optimize the HAProxy load balancer for extreme usages
  with CPU usage at 85% system, you focus mostly on network stack and drivers
- Develop tools to stress test haproxy and network stack
  ⇒ *focus on efficiency and nothing else*
- Some recreation with easier tasks like GPIO, I2C, watchdogs, leds, LCD drivers for our ALOHA appliances

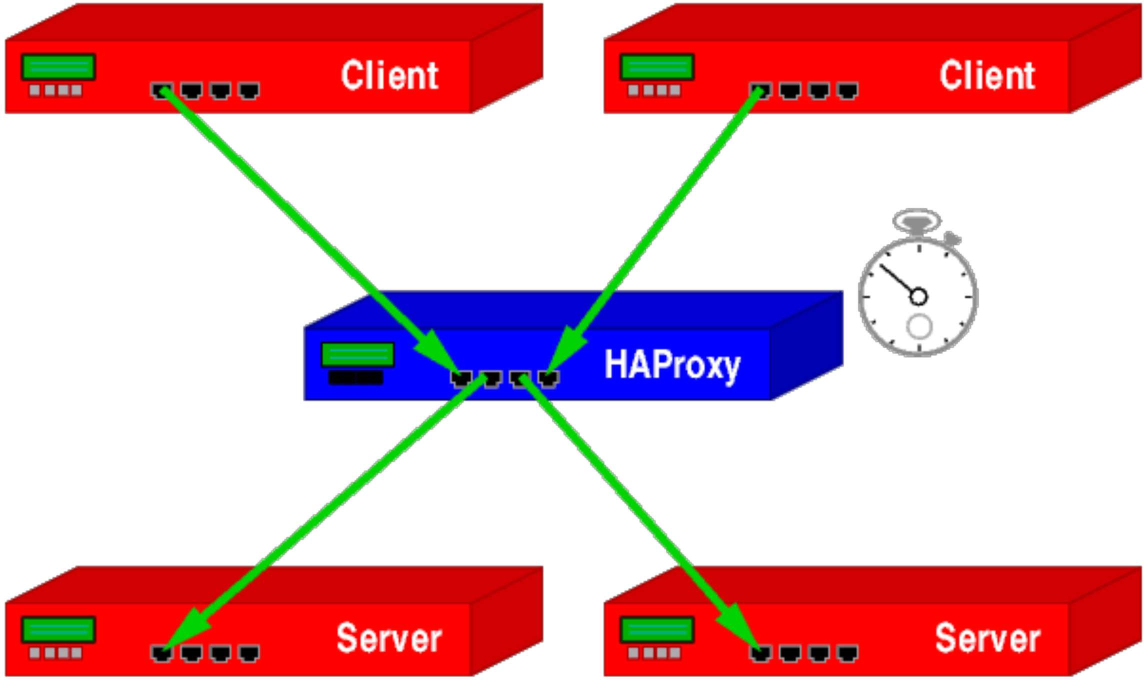
Developing network products

- Always requires multiple machines, minimum 3 :
    - the tested system must be saturated
    - the testers must not be saturated



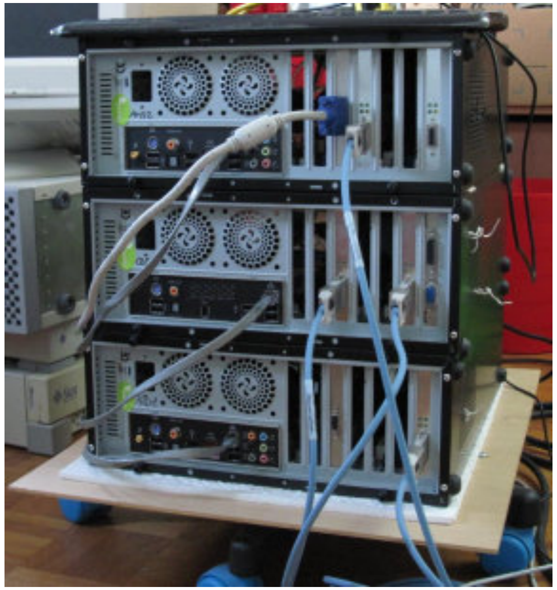
Developing network products *(cont'd)*

- CPUs and software become faster ⇒ hardware limits are hard to reach
- Preferrably use 5+ machines (&switches) for reliable 20-60 Gbps tests



Developing network products *(cont'd)*

- Cost and heat of hardware plus switches means they have to stay in the lab

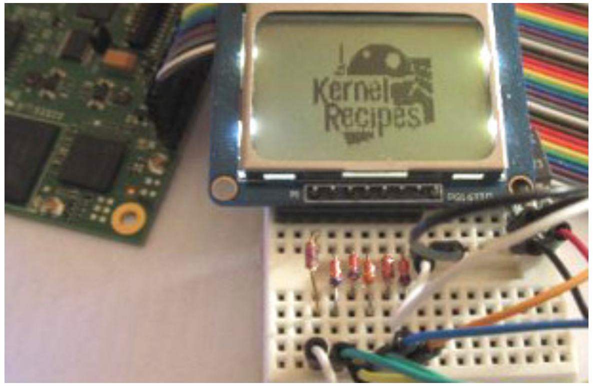


- Testing young features may require to update all kernels for each test (eg: TFO)

  ⇒ **All this just to stress your code, how to simplify this ?**

VMs ? *not really suited for these tasks*

- Network workloads are highly sensitive to latency and jitter
- Nothing reproducible ⇒ not suited for performance testing
- SMP VMs do not always reproduce race bugs
- Hardware emulation does not reflect reality (eg: bus latency, interrupt rates, inter-packet gaps,...)
- Black-out on what the HV does (extra memory copies, merging, etc...)
- Also VMs are not directly connectable to a breadboard :-)



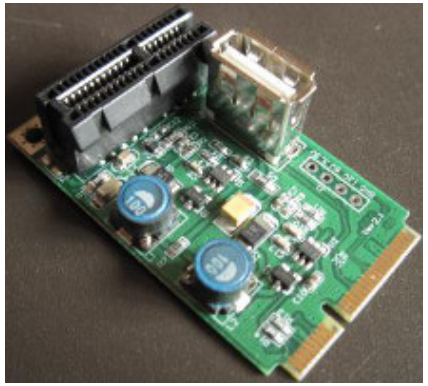
Why not consider small HW alternatives ?

- Portable code does not depend on specific hardware
- Various architectures available: x86, MIPS, ARM, ...
  don't be afraid of cross-compiling!
- Smaller hardware is easier to saturate
  ⇒ less network nodes needed
- Response time is emphasized (good for user interfaces)
  ALOHA's WUI and APIs are developped on ALIX
- Good reproducibility, sometimes even better than desktop PCs
- Small caches / memory busses tend to magnify the impact of memory accesses
- Single/dual issue CPUs, low GHz, 1..few cores, emphasizes effects of missed optimizations

⇒ **Small HW behaves much like large HW with all numbers scaled down**

Pros: feels much more "real" than VMs

- Large choice of SMP hardware where concurrency really matters
- Low latency access to onboard devices
  ⇒ most of the time is spent in **YOUR** code
- Availability of Gigabit Ethernet ports, compatible with laptops/desktop PCs
  ⇒ 4 GigE ports on the AX3-4 :-)
- Availability of a wide choice of busses (PCI, PCIe, USB, I2C, SPI, ...)
  ⇒ develop/experiment with standard hardware then move it to the target



Pros: Cost

- I can afford to buy the hardware I want to play with, and have as many as I need (eg: 4-5 ALIX at home, ...)
- I don't fear frying one (think GPIO, overclocking, stopping fans)
  ⇒ never fried one even after some soldering
- I don't care about the risk of damage when carrying them in by bag
- Sometimes the price/features ratio is attractive even for serious projects (eg: BBB)

Board	Price	Main features
BeagleBone Black	$45	Cortex A8, HDMI, LAN, USB-powered
PC Engines ALIX 2D13	~$100	x86, 3 LAN, mini-PCI, IDE
GlobalScale Mirabox	~$150	ARMv7, 2 GigE, PCIe, USB3

Some manufacturers are willing to donate hardware when you ask them!

Pros: Silence / Heat / Space / Weight / Power

- Only fanless accepted, rare tolerance for very slow fans (Atom, XP-GP)
- Heat : limited heating is the condition for no fan
- No fan means no dust
- Space : no dedicated room, it's easy to stack them on the desk
  order short cables, but use power adapters with long cords
- Not a single hard disk anymore at home nor in the lab
- Weight : bring a few machines everywhere with you to test your ideas as they come.
- Power : you don't want to carry the power adapters with you, and prefer to power the devices over USB (LS/Dockstar/BBB only).

Pros: Build time

- Kernel: start from xxx_defconfig and disable unused features
  (eg: do you need SOUND ? WIRELESS ? BT ? SATA ? IPSEC ? IPV6 ? NETFILTER ?)
  ⇒ stripped down configs build in just a few tens of seconds on a core i5 laptop
- Incredibly appreciated for "git bisect" (make clean/make) !
- But no savings to expect on user land in general

Choose the compiler version and/or options that provide you the best balance between features and build speed. Remember this is **not** the target platform.

Pros: Boot time, console access

- PC: one minute from power on to prompt is not uncommon (worse with PXE)
- Booting in 10 seconds is normal on small hardware.
- Can do better for some usages using a minimal initramfs
- Keep a list of very short copy-pastable boot commands
- Boot loaders could be improved but are already great (everything editable)
- Even the nastiest kernel panics are caught verbatim over the serial console

Pay attention to the console, must be accessible from outside, and use true RS232, 3..5V TTL or self-powered USB.

Pros: Resistance to bad behaviour

- No risk of frying a hard drive or losing BIOS settings with power cycles.
- Low currents, much more resistant than PC hardware to soldering mistakes
- Does not require any special packing even in a bag full of junk
- **BUT! be careful not to confuse 12V adapters with 5V ones, this can be fatal!**

Pros: Wide availability of hardware settings

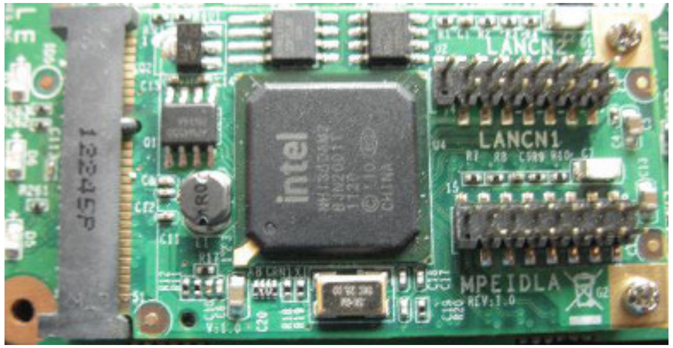
- Sometimes you want to understand what limits your code
- When you have the datasheets (*still waiting for your response Marvell*), you realize how much margin you get on certain settings such as CPU speed and RAM latency.
  Dockstar's RAM bandwidth may be doubled by setting registers
- Various other things such as PCIe speed, component priorities, etc...

  ⇒ Helps figuring what your code is sensible to.

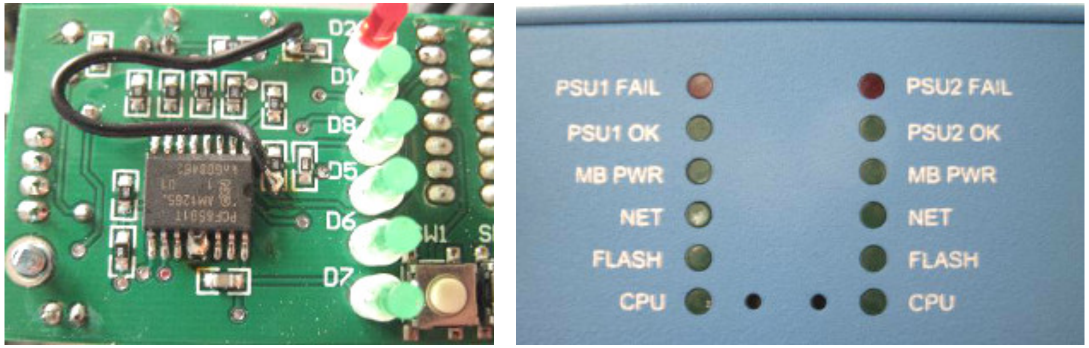
Pros: options for supporting standard components

- USB is omni-present

  Several devices offer PCIe busses
  ⇒ appreciated to run a PC NIC with the same driver as the target (eg: igb runs pretty well on the mirabox) →



- I2C is almost always available and usable to develop various I/O boards



Prototype of ALOHA rack duo power distribution board

Cons

- ASM optimizations or extended instruction sets not always present (eg: bitops, SSE missing on Atom, integer divide missing on many ARM CPUs, ...)
- Hardware perf counters can differ or not exist, making profiling harder
- Timing-dependant bugs often require the target system
- Not all available boards are fully supported by mainline kernels, be careful
- Some boards are hard to boot (complex cmdlines with many memory mappings, out of tree drivers, such as the snowball ...)
- Some boards have limited designs that can fool you into believing there is a problem with your code (eg: 470 Mbps Gig NICs, Ethernet over USB, ...)
- Some platform specific patches might not be available for your platform (eg: Netmap)

  ⇒ **Using small platforms does not save you from testing on the target platform from time to time.**

Important

- Don't waste your time trying to build your own toolchain : use **crosstool-ng** instead (and if it fails, contribute back)
- Don't waste your time trying to get your preferred development platform to work, use another one instead.
- Don't try to establish performance ratios between your development platform and the target. **That never works!**
- Don't try to optimize for the development platform, think about the target
- Do not claim any big improvement before verifying it on the target

General hints

- Prefer **little** endian and alignment traps to ensure maximum portability
- Minimal kernel to save both build time and boot time
- Minimal initramfs to save on boot time
- Put your SSH pubkeys in the root account
- Configure .ssh/config to connect using short names (eg: ssh c1/c2/s1/s2)
- Ensure that the target rootfs has **all** the required tools
- NFS root can be even better but limits network experimentations
- Don't be ashamed to keep git repos with extremely ugly patches that will never reach mainline if they save you some time and hassle.
  ⇒ Only issue is with git bisect.
- And don't pollute upstream maintainers with your specific crap !

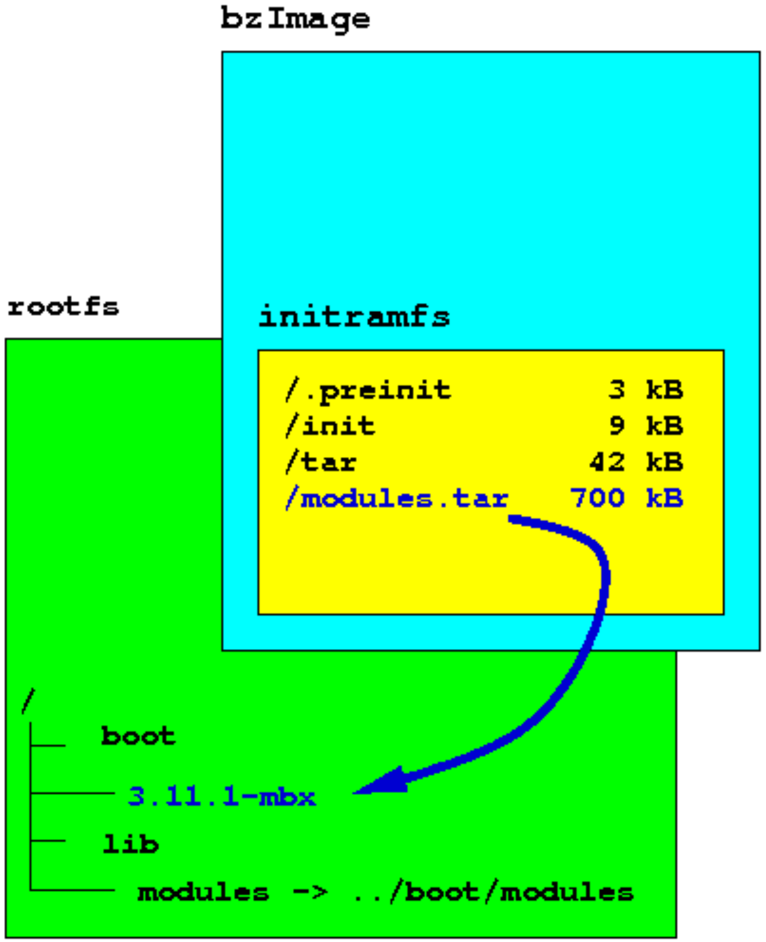
Kernel specific hints

- `make ARCH=$arch CROSS_COMPILE=$cross_prefix {oldconfig|zImage|modules}`
- Enable CONFIG_EARLY_PRINTK, CONFIG_IKCONFIG_PROC, CONFIG_DEVTMPFS_MOUNT
- Prefer modules for your driver, but some drivers might have annoying side effects
- Better have several light configs than a single fat one
- Base your work on mainline and always keep one recent working version
- Keep **all** your experimental kernel configs, kernel versions are not enough
- Dissociate the kernel from the rootfs
  have the kernel embed its own modules. Busybox in an initramfs also helps a lot.
- A build script does everything and even appends DTB for multiple boards and uploads the kernels to the TFTP area.

Kernel specific hints (cont'd)

Overview of my kernel build script

1. make prepare
2. make -j modules
3. make modules_install
4. tar cf initramfs/modules.tar $INSTALLDIR
5. make -j {bzImage|uImage|zImage}
6. optionally: make dtbs && mkimage



Userspace hints

- Spend enough time optimizing the boot process, it will pay off very quickly.
- Disable all unneeded services, even the rarely used ones
- Disable anything that requires network connectivity
- Rootfs may appear in various forms (RAM, CF, NAND, MicroSD, USB, NFS, ?)
- Use fast passwords (if any) for the console, and none for sudo
- Keep a local config if booting many similar machines over the network
- Don't use the same IP address on multiple devices
- And note MAC/IP/hostname somewhere for easier identification when you start to get lost (/etc/hosts, DHCP configs, ...)

Userspace hints (con't)

- Do not miss any single tool, install any required library etc...
  ⇒ you need to **test**, **debug** and **measure** your code!
- Mandatory in the rootfs : strace, taskset, top, vmstat, perf, tcpdump, if_rate
- Mandatory available : gdb (both native and cross)
- Copy-pastable notes with setup commands, local scripts, ...
- Sometimes useful to have some generic scripts that depend on the device they're running on (eg: network setup) .
- If playing a lot with the network, reserve an Ethernet port for admin if possible

Better, build your own rootfs!

You need :

- /sbin/init : busybox is a good start for this

  having a pre-init is even more convenient
- /dev : use devtmpfs
- /proc, /sys: mount them as usual
- /bin/everything : busybox is fine here as well

⇒ My rescue rootfs contains **`dev, tmp, bin, sbin, lib, var, usr/bin, usr/sbin, proc, sys`** and starts with a pre-init calling **`/busybox --install`** when mounting root fails.

Various other projects allow you to build fully functional rootfs in the 1..10 MB range (OpenWRT, Buildroot, LFS, OpenEmbedded, ...)

Various boot choices

- No one size fits all
- Kernel options
  - Local storage : FS (grub, syslinux), NAND (u-boot, redboot)
  - Remote storage : PXE (pxelinux), TFTP (u-boot, redboot)
- Rootfs options
  - Local storage : block device (ext2/3/4, squashfs, ...)
  - Local storage : MTD (eg: ubifs)
  - Remote storage : NFS
  - Loaded into RAM by boot loader from local/remote image (ext2/3/4, squashfs, ...)
  - Large initramfs embedded in the bzImage

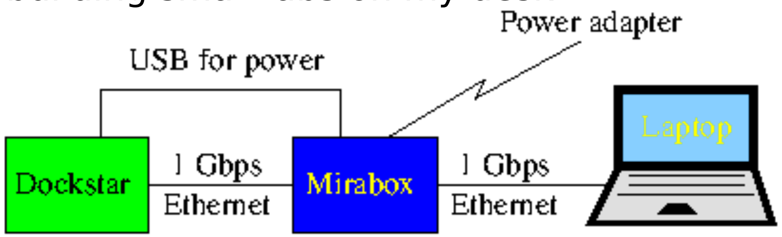
Loading over true GigE is generally faster than local devices which are generally faster than 100 Mbps network.

Hardware suggestions

My platforms of choice at the moment :

- Dockstar for GigE powered over USB *(modded)*
- OpenBlocks AX3-4 for SMP and high performance networking
- Mirabox when PCIe is needed, or to have two GigE ports in my pocket
- ALIX for i586, except I2C (unreliable)
- Atom N450 for x86_64 (but has a small **fan**!)
- GuruPlug for I2C devs *(modded)*

⇒ Various combinations when building small labs on my desk



My ideal board would be the size of a BBB, have a dual-core Cortex A9, 2 true GigE ports, a few I/Os and be powered over USB.

Conclusion

Developing on the kernel is easy when starting with hardware drivers, and can be really cheap thanks to the wide choice of platforms. Everyone can hack in the kernel.

My everyday bag's contents