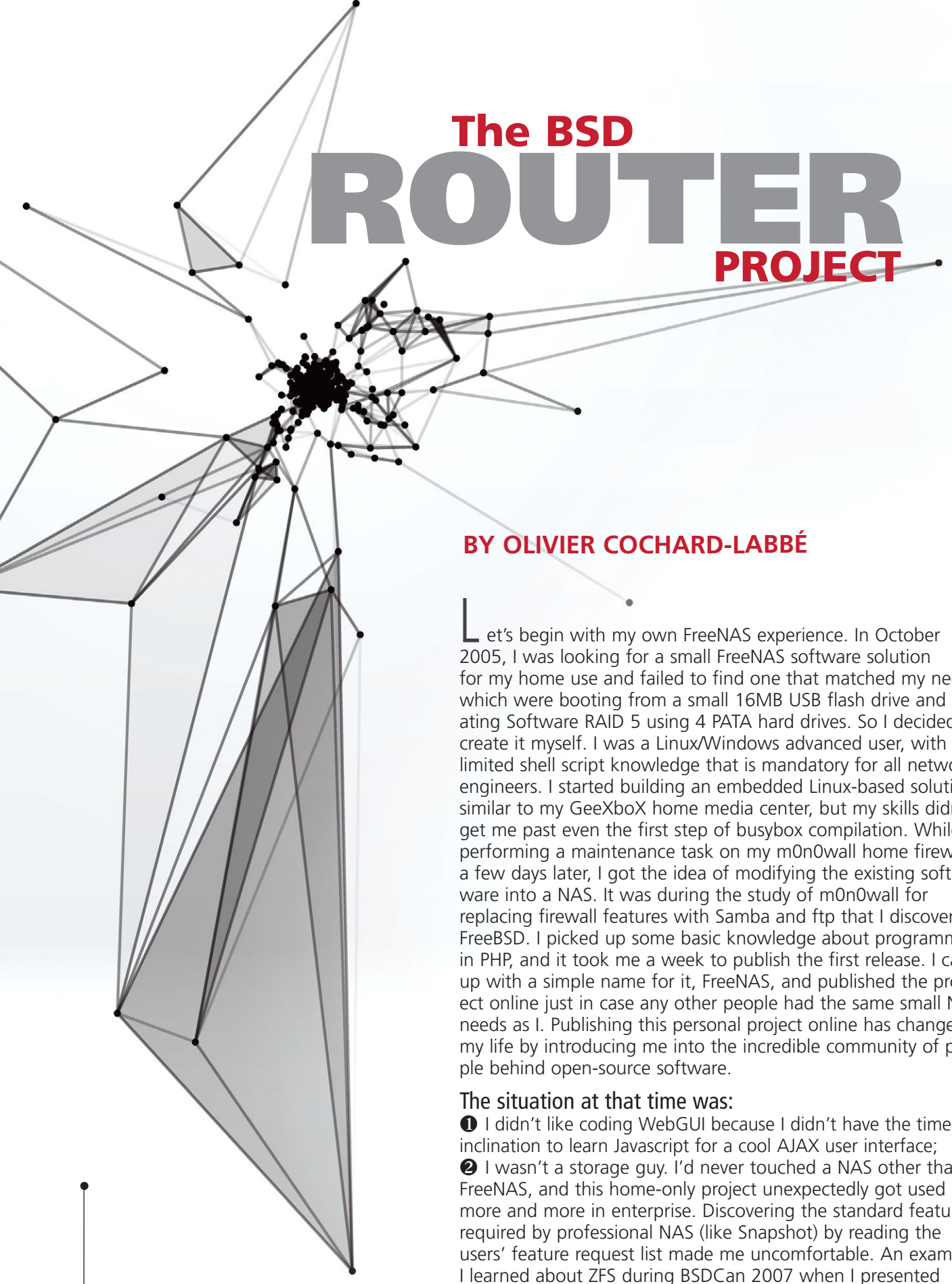


The BSD ROUTER PROJECT

BY OLIVIER COCHARD-LABBÉ

Let's begin with my own FreeNAS experience. In October 2005, I was looking for a small FreeNAS software solution for my home use and failed to find one that matched my needs, which were booting from a small 16MB USB flash drive and creating Software RAID 5 using 4 PATA hard drives. So I decided to create it myself. I was a Linux/Windows advanced user, with the limited shell script knowledge that is mandatory for all network engineers. I started building an embedded Linux-based solution similar to my GeeXboX home media center, but my skills didn't get me past even the first step of busybox compilation. While performing a maintenance task on my m0n0wall home firewall a few days later, I got the idea of modifying the existing software into a NAS. It was during the study of m0n0wall for replacing firewall features with Samba and ftp that I discovered FreeBSD. I picked up some basic knowledge about programming in PHP, and it took me a week to publish the first release. I came up with a simple name for it, FreeNAS, and published the project online just in case any other people had the same small NAS needs as I. Publishing this personal project online has changed my life by introducing me into the incredible community of people behind open-source software.

The situation at that time was:

❶ I didn't like coding WebGUI because I didn't have the time or inclination to learn Javascript for a cool AJAX user interface;
❷ I wasn't a storage guy. I'd never touched a NAS other than FreeNAS, and this home-only project unexpectedly got used more and more in enterprise. Discovering the standard features required by professional NAS (like Snapshot) by reading the users' feature request list made me uncomfortable. An example: I learned about ZFS during BSDCan 2007 when I presented FreeNAS for the first time, and I'm still not convinced that using

such a complex file system for home use makes sense.

❸ I had a full-time job (network engineer). The work on FreeNAS was just a hobby, but the time spent managing the unexpected success of the m0n0wall patch forced me to reconsider my involvement in the project because it consumed a lot of my free time. Also, I wasn't able to provide a clear road map as to which direction to take the storage technology, and I was not able to understand some user problems like Unix file permission mapping with CIFS.

My objective was to keep FreeNAS a NAS and not become a generic server offering print or bittorrent client services, but I understood the desire of users to have these features. The solution was to add plugins, keeping the base core feature offering NAS, but allowing users to transform it to a generic server if they so desired. At that point, I considered a full rewrite of FreeNAS as the m0n0wall base—which used an mfsroot—didn't allow for the easy addition of plugins. It was during this time that I discovered nanoBSD, but my limited free time did not permit me to do it myself (children are the worst enemy of an open-source project managed during one's free time!). Internal team discussions focused on a new base for the next FreeNAS and when the main developer (as it was not me during this period) published the idea of using a Linux Debian base for the new release, it created an unexpected buzz online. Reacting to this buzz, iXsystems contacted me and offered their help with the development. I decided to give them the full project and for free, because I've always tried to keep money far away from my hobby.

So after being liberated from FreeNAS in December 2009, my idea was to start a new project, but this time in a domain I knew more about. It would be a software router, based on FreeBSD, because that was the only OS where I felt confident.

My objectives were:

❶ To not target the home user as there were already some WebGUI firewalls like m0n0wall (replaced by the t1n1wall or SmallWall today) or pfSense (or OPNsense) for that.

❷ To use nanobsd as the base as I hate reinventing the wheel. nanobsd is a great tool for building FreeBSD-based appliances.

❸ No WebGUI. I simply can't imagine how a router configuration can be represented inside a GUI. And my previous bad experience with the "graphical" Nortel Networks Site Manager software for configuring AN/ARN/ASN routers didn't help me either. I think it's easier to just manage a text file. This "feature" helps to keep the home user away from the project as well.

❹ Configuration management. At the start, adding NETCONF support was a consideration, but its incredible complexity (more than 20 RFCs) and its slow deployment made me reconsider and instead opt for standard and well-known tools like Ansible.

❺ I believed in software routers (before 2009), but felt something was wrong with the slow software forwarding performance of generic x86 servers. Their powerful CPU and NIC didn't match with their very slow forwarding rate, and a few years later, netmap confirmed that it was just a software problem.

THE BENEFITS OF USING BSDRP

BSDRP is a customized nanobsd disk image targeting router usage. This is standard FreeBSD that has the same behavior as a network appliance and is accepted by network administrators.

- Upgrading the system is like an old Cisco IOS. You just install the new firmware file and reboot it, thanks to the FreeBSD POLA (Principle of Least Astonishment) that allows you to upgrade the system without major changes to the existing configuration files.
- The read-only mode of the file system allows you to power unplug/replug appliances without problems.
- Small size. A 512MB flash drive is enough for BSDRP. The zipped upgrade file is about 40MB.

By default the nanobsd image build script permits you to add existing packages to the final nanobsd image. BSDRP uses a highly customized nanobsd configuration file that allows you to build ports (with specific compilation options) during the nanobsd image generation. This feature allows cross-compilation of an i386 image from an amd64 host.

BSDRP images disks are published for i386 and amd64 architectures, but the nanobsd script was improved so as to allow sparc64 images as well. However, since the death of my last sparc64 server, I've stopped publishing sparc64 images.

The current selection of packages includes:

- Quagga and Bird as unicast routing daemons; the first is for old Cisco users, and the second is the next-gen routing software;
- mrouted, pimd, and pimdd as multicast routing daemons;
- native carp, ucarp, and freevrrpd for high availability. The presence of ucarp allows you to

use carp on some interfaces, and freevrrpd on others;
❖ native netgraph netflow and pmacct for IP accounting. Enabling netgraph negatively impacts the forwarding performance, which is why there is pmacct;
❖ mpd5 for all advanced PPP protocol support;
❖ OpenVPN, ipsec-tool for IKEv1, strongswan for IKEv2;
❖ Python, because this language became more and more used by network admin teams. This also allows BSDRP to be managed by Ansible;
❖ Exabgp, a python-based tool—the swiss army knife of BGP routing;
❖ iperf and netmap's pkt-gen, for benchmarking;
❖ ISC-DHCP server and dhcprelay;
❖ tayga, a userland stateless NAT64 daemon;
❖ and monit as a process monitor.

Some specific tools and script were written for this software:
❖ config—allows you to save/rollback/send/receive system configuration;
❖ system—allows upgrading/checking the integrity of the system;
❖ tuning—experiments with collecting system data (number of CPUs, model of NIC, etc.) and proposes a tuned value for obtaining the best forwarding performance;
❖ equilibrium—helps to bench VPN configuration (IPSec, OpenVPN, etc.);
❖ quagga-bgp-netgen—a very simple BGP route generator using quagga;
❖ and some tcsh command completes specific to a router.

A Testing and Documenting Use Case

Once the BSDRP image was generated, I had to build a network lab, but for building a network you need multiple routers. I wrote some shell scripts for different hypervisors (bhyve, virtualbox and qemu) that allow the easy generation of a full mesh network with multiple routers in one command line. My main labs are done with bhyve because of the incredible speed of the VM. But the virtIO NIC presented by bhyve didn't support ALTQ, so I had to use VirtualBox which can emulate a virtual NIC compliant with ALTQ.

The idea is to publish a lot of network lab examples—including their full configurations—on the BSDRP website, but at the moment, only a few examples are available (BGP, OSPF, Multicast, VRRP, PPP, GRE, GIF, IPsec).

Benchmarking Forwarding Performance

BSDRP should be tuned by default to deliver the best FreeBSD forwarding performance. But how do you tune FreeBSD for router usage? There are many "tuning guides" online, but the majority of them give magic values without explaining why some values are better than others. My idea was to focus on a specific variable and to test a range of different values against this variable. But for that, I had to learn how to do a correct bench, because during this step I learned that even for an experienced network guy, doing benches is a complex task.

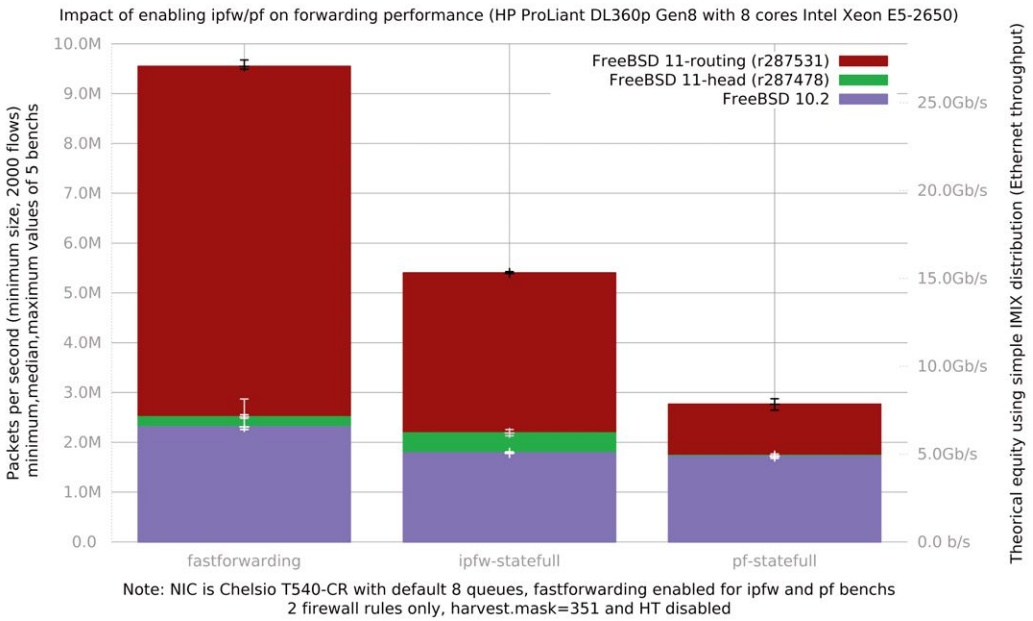
What are the main parameters for benchmarking a router? Hopefully there are some RFCs explaining how to measure their routing performances like RFC: 1242, 2544, 3222, and 5180. But when it comes to measuring more advanced features like IPSec/GRE tunnels, there is not an official guide.

The bench methodologies were adapted to my "end user" usage. I'm not a hardware vendor nor a routing software seller, so I don't really care about testing with multiple frame sizes or presenting only the "best" profile as firewall vendors do when presenting their outstanding firewall performance in Mb/s…with only Jumbo Frame. I'm interested only in the "worst" scenario which means to bench using only minimum frame size. This means that a lot of designations of my benches can be renamed as "number of packets forwarded during a Denial-Of-Service."

But the RFC didn't give complete details for doing a bench. For example, RFC2544 requires multiple trials. But how many trials, and what should we do with multiple trial results?

The answer came from reading the FreeBSD mailing-list: a lot of bench results published on the mailing-list are through the ministat utility. Ministat calculates the fundamental statistical properties of the trial results: minimum, maximum, median, average, and standard derivation. This provides the first answer to the question of "how many trials?" Three at minimum, but more is always better. On my benches, because my device (DUT) is rebooted between each trial, I'm limited by the incredibly long BIOS POST start time of the commodity server. For a small 30-second trial, it can lose about 4 minutes for booting my HP or IBM servers (and my benches often need more than 200 reboots). So I limit my number of trials to 5 when I'm benching a slow BIOS POST server, and increase it to 10 trials for a rapid BIOS POST server (like PC Engine APU or Netgate's RCC-VE appliance).

This means that if you see any kind of "benches" online that present just one measured value (without derivations or number of trials), you can ignore them. Once the input was collected—parameters to bench, methodology, number of trials—I wrote a shell script to automate the bench. Presenting the final results was not easy, as I needed to represent the errorbar concept on the final graph, and more importantly find a good title. As an example, my classical "server resumed performance" graph shows three bars: one is fast-fowarding, a second with IPFW enabled, and the last with PF enabled.



Some readers wrongly interpret these results as showing that IPFW is a better firewall than PF because it has a better packet-per-second (PPS) value. But that is incorrect. Benching a firewall is part of a totally different (and more complex) world, and hopefully a firewall is not reduced to its PPS performance. I had to use a long title on my graphs to avoid this kind of interpretation. And for people interested in the impressive value displayed under the name "FreeBSD 11-routing," it's Alexander V. Chernikov's projects/routing available on the public FreeBSD svn.

Publishing results online is always challenging because they will be criticized, and you have to be sure they don't contain errors. The last thing I want to do is waste a developer's time working on a nonexistent problem. Hopefully my early errors were quickly pointed out by the community. As an example, at the beginning I was generating just one IP traffic flow, which didn't allow the use of all the multi-queue features of NIC. A second error was in not disabling the autoscale feature of graphing software, as the resulting graphs then artificially increased the difference of the results.

The final problem in presenting data is the unit. The main performance value of a router is the packet-per-second unit, but regular users are expecting a maximum bandwidth in Mb/s (or worse: MB/s). Because my benches are using only the smallest packet size (64B), it's possible to use the simple Internet Mix (IMIX) distribution for an estimated equivalence in Mb/s. Even if the simple IMIX reference (58% of 64B packet, 33% of 570B packets, and 8% of 1518B packets) didn't match current distribution that is more bimodal (40% less than 100B and 30% more than 1500B in 2012), using the simple IMIX allows us to obtain values constant in time.

Now that I've defined my methodology, I need to add more benching features like IPSec/OpenVPN or a way to test forwarding performance vs. number of routes.

What is the EINE Sub-project?

In 2014, my employer, Orange, asked me to build a proof-of-concept allowing deployment and management of any kind of x86 network appliance over the Internet with minimum administrative tasks. I reused BSDRP for this task and created a sub-project called EINE for "Easy Internet vpN Extender." This unique nanobsd image once installed can be configured for different roles.

Manager. It's the host that stores all appliance configuration parameters and Ansible playbooks for all roles (as a plain text file).

VPN Gateway. To terminate client routers VPN tunnels.

End-clients that can be VPN-Wifi-routers, serial terminal servers, captive portal appliances, etc. It's the default configuration of the nanobsd image configured for plug&play. Plug&play is done by an Internet facing NIC configured in DHCP client mode and with an openVPN client using a generic certificate connecting to the VPN gateways. When the generic certificate is used, the appliance is in an "unregistered" state and all traffic

from it is denied. The appliance needs to receive its configuration files (including certificates) from the manager in order to change to a "registered" state.

All administrative tasks are done from the manager that is using simple helper scripts (in Python) and Ansible.

Typical tasks include:

- ❖ Displaying lists of unregistered appliances connected to all VPN gateways
- ❖ Applying a role to an unregistered appliance (it's a simple Ansible group mapping)
- ❖ Upgrading all connected appliances
- ❖ Deleting a registered appliance that was declared stolen

Notice that I've chosen to use a FreeBSD -head for this project for two main reasons:

❶ To help FreeBSD by testing -head;
❷ To force myself to learn how to build continuous integration servers, and to write tests (because -head is so useable I didn't take time for this task).

Planned Features

The next major change will be to test a solution other than nanobsd. I'm using a highly customized nanobsd script that allows us to build, port, or compile some of the /usr/src/tools during nanobsd image generation. But the FreeBSD port system changes a lot and following its evolution takes a lot of time. Recently poudriere added the option of generating images and firmware in nanobsd. Using this feature will hide all port system changes behind the powerful poudriere.

The second major change will be to do more tests on FreeBSD projects/routing and its stability. I will eventually switch from the release branch to this one. And last but not least, I'm eager to test the new netmap-forwarding software (announced during BSDCon Brasil 2015)! •

OLIVIER COCHARD-LABBÉ has 16 years of network engineering experience. He discovered FreeBSD by accident in 2005 while patching m0n0wall to add NAS features. Since then he has contributed to FreeBSD mainly by focusing on networking and maintaining simple ports. Trying to convince his colleagues that open-source software on generic x86 servers is the future is his favorite pastime at work. Rugby, running and freediving are his sports. He lives near Nantes, France, with his wife and two young daughters (whom he tries to keep away from video games and TV).