Porting dummyet to Linux and Windows (and userland)

Luigi Rizzo, Università di Pisa

May 14, 2010

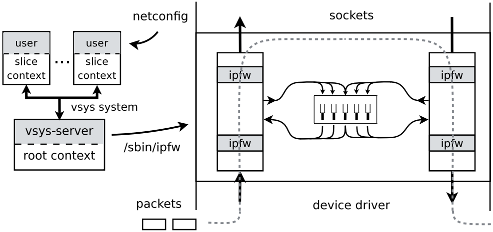
Summary

In this talk we will describe the issues and lessons learned in porting a network-related kernel module from FreeBSD to different operating systems.
In detail:

- motivation and objectives;
- description of the system being ported;
- porting strategy;
- identification of the subsystems involved;
- system-specific issues;
- lessons learned.

Motivation for this work

As part of the ONELAB2 project **www.onelab.eu** we needed to implement in-node emulation for PlanetLab.



We opted for a port of **ipfw** and **dummynet** because:

- existing Linux solutions (tc+netem; NISTnet; netpath) were not as flexible as dummynet;
- a non-negligible integration and porting work was still needed even with the above systems;
- a Linux port was desirable in itself.

Objectives

During the work, we decided to address the following issues:

- add scheduler support (direct requirement of the Onelab project);
- improve scalability (fixes a performance issue in Onelab);
- provide some user-level testing tools (ease development, improve the quality of the software);
- create a generic Linux port, because Planetlab nodes use different Linux versions;
- develop OpenWRT and Windows versions, as it only required a limited additional effort, and would make the tool available to a much larger user base.

The systems to be ported

Ipfw and Dummynet **info.iet.unipi.it/∼luigi/dummynet/** are a firewall and traffic shaper/network emulator, made of:

- a user interface, **/sbin/ipfw**, running in user space and communicating with the kernel through a control socket;
- several kernel modules (**ipfw.ko**, **dummynet.ko**, schedulers...) attached to the **pfil** hooks to intercept packets.

The original code was not specifically designed for portability, so it uses several FreeBSD-specific structures and subsystems:

- mbufs, pfil hooks, memory allocator, locking, timer services;
- ip_output() and netisr_dispatch();
- routing table, module management, control sockets.

Porting approach

Our approach was to port the code to Linux with as little modifications as possible to the original code:

- faster, less error prone;
- easier to keep the software up to date;
- small performance loss is not a concern.

Workplan:

- identify differences among platforms;
- provide replacements for headers;
- provide wrappers for similar functions/subsystems;
- develop glue code to map FreeBSD kernel APIs to underlying OS APIs.

We do not require nor use any GPL code.

Porting the userspace code

Porting the userspace code to Linux/Cygwin was almost straightforward:

- language and APIs are relatively portable across platforms (BSD, Linux, Cygwin);
- no strange linker tricks in the code;

Main points:

- header adaptation – discussed next;
- missing library functions (`humanize_number(), ...`) obtained from the original, BSD-licensed source code;
- Windows: remap `setsockopt()` to `DeviceIoControl()` (similar in principle, device handle instead of a socket);
- `sysctl` emulated over the control interface;

All extensions are in one file, `glue.c` – 800 lines (mostly for library functions and sysctl emulation).

Building userspace code with Windows tools

Windows porting with native tools (MSVC, tcc) is slightly more difficult:

- useful because it produces a GPL-free binary;
- larger differences in headers, APIs and basic data types (WORD DWORD FAR ...);
- missing functionalities (fork, process control, printf formats ...);
- missing compiler features (e.g. C99 initializers in MSVC).

Problems solved with some headers tricks, minor rewrites or removing some functions: only two small `"#ifdef TCC ..."` sections in ∼7000 lines of code.

Porting the kernel side

Porting the kernel code is much more challenging and interesting:

- lack of cross-platform standards, both for header names and content, and kernel APIs;
- many more subsystems involved.

Header remapping and large use of macros go a long way in reducing differences.

Step 0: userspace version of the kernel code

We started by building a userspace version of the kernel code:

- quickly identify missing headers and libraries;
- experiment with various porting approaches.

Not a wasted effort:

- eventually, we had a daemon that could talk to `/sbin/ipfw` through emulated `*sockopt()`;
- useful to test rule injection and listing;
- opened the way to develop the scheduler testing code;
- we plan to add packet handling (e.g. from a PCAP file) to test packet matching functionality and performance.

Step 1: Header remapping

There are significant differences in kernel headers:

- some BSD headers are missing on other systems;
- some have the same name but different content;
- some have different names for a given content;

From many headers we need only a handful of lines, so:

- `-include ...` to import common definitions (2 files, ∼1000 lines);
- a subtree `-Iinclude/` contains ∼30 headers copied almost verbatim from FreeBSD;
- a subtree `-Iinclude_e/` is populated with ∼50 empty headers, for files with no (remaining) content.

Kernel compile flags start with

```
-nostdinc -include ../glue.h -include missing.h
-Iinclude -Iinclude_e ...
```

Header contents

The -include'd headers do a variety of remapping tricks:

```
#define ifnet                    net_device      /* remap */

#define printf(fmt, arg...) printk(KERN_ERR fmt, ##arg)
#define bcopy(_s, _d, _l)   memcpy(_d, _s, _l)

#define IP_FW_SETSOCKOPT \
        CTL_CODE(FILE_DEVICE_IPFW, IP_FW_BASE_CTL + 1, \
        METHOD_BUFFERED, FILE_WRITE_DATA)

#define _SYSCTL_BASE(_name, _var, _ty, _perm)        \
        module_param_named(_name, *(_var), _ty,      \
                ( (_perm) == CTLFLAG_RD ) ? 0444: 0644 )
```

Most of these macros are the result of a comparison of how the various subsystems are implemented on different platforms.

Step 2..N: handle kernel subsystems

An interesting part of the work has been identifying the differences in various subsystems:

- packet representation and packet hooks;
- memory allocation;
- locking;
- timers (API and resolution);
- module support;
- userland/kernel communication;
- OS-specific issues.

These will be described in the next slides.

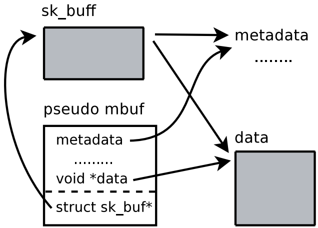
Packet representation

In-kernel packet representation always uses a descriptor to store metadata and a linked lists of buffers:

- mbufs on FreeBSD;
- skbufs on Linux;
- NDIS_PACKETs on Windows.

Our code uses mbufs, so we do the following:

- create mbuf lookalikes on entry;
- copy metadata from native representation;
- reference or copy data;
- destroy the wrapper on exit.



Exact details depend on packet hooks behaviour.

Packet filtering hooks

Dummynet must sometimes hold/delay/drop packets. Slightly different semantics among systems:

FreeBSD **pfil hooks** allow a hook function to free or hold a packet;

Linux **netfilter hooks** require all packets to be marked and returned. Packets can be held on a subsequent QUEUE call;

Windows **NDIS miniport** modules do not allow modifications to packets. A module must replicate the packet to hold/modify it.

Some unnecessary data copies could be saved if FreeBSD had a clear separation between classification and action on the packet.

Memory allocation

`malloc()` remapped to OS-specific allocators:

- `kmalloc()/ kfree()` on Linux;
- `ExAllocatePoolWithTag()/ExFreePool()` on Windows;

UMA allocators are replaced by a much simpler version:

```
typedef int uma_zone_t;            /* the zone size */
#define uma_zcreate(name, len, _3, _4, _5, _6, _7, _8)  (len)
#define uma_zalloc(zone, flags) malloc(zone, M_IPFW, flags)
#define uma_zfree(zone, item)   free(item, M_IPFW)
#define uma_zdestroy(zone)      do {} while (0)
```

Locking

Fortunately we use a very simple locking mechanisms (rwlocks, rmlocks, mtx).

- define/declare/lock/unlock/destroy wrapped in macros;
- map to `spinlock_t` on Linux;
- map to `FAST_MUTEX` on Windows.

Timers (and callouts)

Used for two purposes:

- Return the time of day, with $< 10\mu s$ resolution and precision.
  - getmicrouptime() or microuptime() on FreeBSD;
  - do_gettimeofday() on Linux;
  - custom replacement (TSC-based) on Windows as we could not find a function with less than 10ms resolution.
- Wake me up after time T:
  - callout_init/callout_reset on FreeBSD.
  - mapped onto init_timer()/add_timer() on linux;
  - Deferred Procedure Calls (DPC) on Windows: KeInitializeDpc()/KeSetTimer()
  - hardest part was *locating* the right API to set the kernel tick on Windows (ExSetTimerResolution()).

Module support

Modules have descriptors to indicate constructors, destructors and dependencies:

```
DECLARE_MODULE(dummynet, dummynet_mod,
        SI_SUB_PROTO_IFATTACHDOMAIN, SI_ORDER_ANY-1);
MODULE_DEPEND(dummynet, ipfw, 2, 2, 2);
DECLARE_MODULE(ipfw_nat, ipfw_nat_mod,
        SI_SUB_PROTO_IFATTACHDOMAIN, SI_ORDER_ANY);
MODULE_DEPEND(ipfw_nat, libalias, 1, 1, 1);
MODULE_DEPEND(ipfw_nat, ipfw, 2, 2, 2);
```

- heavily based on linker sets;
- potential portability issues with different toolchains (e.g. we use MSVC and possibly TCC).

Possible workarounds:

- make the descriptors globally visible;
- manually (or automatically) build the list of module descriptors.

Kernel – userland communication

- `getsockopt()/setsockopt()` on a raw socket.
- Linux has a similar mechanism, slightly different API;
- Windows uses DeviceIoControl(), which operates on a device descriptor;
- in both cases, ported using wrappers to adapt the API;
- the interface has been extended to emulate `sysctl` for platforms missing them.

Linux specific issues and features

- `sysctl` mapped to /sys/module/ entries on 2.6.x, implemented via sockopt on 2.4.x (openwrt);
- jail-id replaced by vserver id;
- IPV6 and in-kernel NAT not implemented yet.

Only one major complaint: **very unstable kernel APIs**.
The code is cluttered by many (~ 30) conditional sections for specific kernel versions;

Windows specific issues and features

- `sysctl` implemented via sockopt;
- no jail/uid/gid matching;
- no matching on interface names;
- IPV6 and in-kernel NAT not implemented yet;
- loopback traffic does not go through NDIS;
- NDIS glue mostly coming from the `miniport` driver;
- installer files available;
- signed kernel modules for 64-bit systems in the works;

Overall porting effort

```
> wc glue.h tcc_glue.h ipfw/glue.c ...

    543     2187    16385 glue.h
    232      884     7141 tcc_glue.h
    841     3051    23538 ipfw/glue.c // sysctl and libraries

    627     2480    18627 dummynet2/missing.h
    547     1802    12914 dummynet2/bsd_compat.c
    906     3681    25957 dummynet2/ipfw2_mod.c

    630     2624    20104 dummynet2/md_win.c
    225      934     7100 dummynet2/winmissing.h

   4551    17643   131766 total
```

Lessons learned

The original code was reasonably portable, despite the lack of any specific effort. Some things could and should be improved:

- need better split between classification and emulation;
- confusion on the endianness of certain fields (`ip_len`, etc.) obfuscates the code and requires writable buffers;
- nested #include would have made header mapping a lot simpler;
- when it comes to locking and other architecture-specific functions, hiding details behind macros is a big advantage.

Availability and Credits

Latest code at http://info.iet.unipi.it/∼luigi/dummynet/
The new code is available for

- FreeBSD HEAD and stable/8
- Linux/OpenWRT
- Windows XP, Windows 7 (32 and 64 bit)
- OSX ? (currently older version. Ask Apple...)

Credits:

- Marta Carbone (Linux port)
- Fabio Checconi (QFQ, KPS)
- Riccardo Panicucci (scheduler API)
- Francesco Magno (Windows port)