NETWORKING SUBSYSTEM CONFIGURATION INTERFACE

Ondrej Lichtner
Master Degree Programme (2), FIT BUT
E-mail: xlicht01@stud.fit.vutbr.cz

Supervised by: Ondřej Ryšavý
E-mail: rysavy@fit.vutbr.cz

Abstract: This paper focuses on the design of a network configuration library with regards to operating system portability. To achieve this portable design, it explores and analyses the currently available network configuration options of Linux and BSD based operating systems and commonly used network devices. The gathered information guides the creation of a portable and extendable library design that will be used to implement a part of the library.

Keywords: Network configuration, Linux, FreeBSD, Ethernet, Infiniband, WiFi Bonding, Bridge, VLAN, kernel interface, library design.

1 INTRODUCTION

The networking subsystem is a vital part of every modern operating system. Correct network configuration respecting performance requirements is the ultimate goal of every administrator. This paper concentrates on individual features of networking subsystems and their available configuration interfaces. Specifically the networking subsystems of operating systems based on the Linux and BSD kernels.

2 CURRENT CONFIGURATION API OVERVIEW

The Linux and BSD kernel interfaces are very similar as both conform to UNIX standards, however there are differences. The information in this section is based on Linux kernel documentation [1, 2] and FreeBSD documentation [3] since it is currently one of the most popular open BSD systems.

2.1 LINUX

For Linux there are several ways a user space application can interact with the kernel and the networking subsystem. The following list contains the most common ones:

- *procfs* and *sysfs* are virtual filesystems, usually mounted in `/proc` and `/sys` that allow the kernel to export internal information to user space in the form of files. They can be read with `cat` or `more` and written into with the > shell redirector. *procfs* is usually used for reading information whereas *sysfs* can be used for both reading and modifying environment properties.
- *sysctl* is a part of the *procfs* filesystem located in `/proc/sys/`. This interface allows user space to read and modify the values of kernel variables. In addition to using the *procfs* the user space can also use the *sysctl* system call.
- *ioctl* system call can be passed a socket descriptor, which can be used to configure the networking subsystem. This is the the most common UNIX network configuration interface.

- *Netlink sockets* are the newest and preferred mechanism for networking applications to communicate with the kernel. Compared to the other interfaces Netlink provides the richest support for communication with kernel components.

2.2 FreeBSD

Compared to Linux the *sysfs* virtual filesystem doesn't exist on BSD systems and there is no equivalent. The *procfs* filesystem does exist, however it can't be used to access the networking subsystem. There are also no *Netlink* sockets on BSD, however, there are *routing* sockets that can be used to manipulate routing tables. On BSD we are therefore limited to using:

- *sysctl* system calls only as the `/proc/sys` filesystem doesn't exist on BSD systems,
- *ioctl* system calls in combination with sockets employed in the same way as on Linux. This is the main interface used on BSD systems and in comparison to Linux systems it supports more features.
- *routing sockets* for configuration of routing tables.

3 NETWORK DEVICE OVERVIEW

Both Linux based and BSD based systems support a wide range of network devices with different use cases. These can be sepparated into two basic categories: *physical* and *software* devices.

Physical devices are network devices that must have a hardware port installed in the computer, their type depends on the hardware and the associated kernel driver. Examples of physical devices are *Ethernet* or *WiFi* network cards.

Software devices do not represent a physical device, instead they exist virtualy within the kernel. This can be useful for a variety of reasons, for example link aggregation (*bonding*) or implementing higher layers of the networking protocol stack (*bridge*).

3.1 Ethernet

Ethernet is one of the most commonly used family of networking technologies used for local area networks. Ethernet devices can be configured using the *ethtool*, *ifconfig* or *IPROUTE2* applications that use the kernel interfaces described in the previous section. The most common configuration options supported on Linux are: changing the MAC address, manipulation of MTU, setting the device speed, network flow classification and others.

4 BONDING

The Linux bonding driver, documented in [2], provides a method for aggregating multiple network interfaces into a single logical *bonded* interface. The behavior of the bonded interfaces depends on the mode, generally speaking, modes provide either hot standby or load balancing services. Additionally, link integrity monitoring may be performed. Bonding interfaces can be configured using either the specific distributions network initialization scripts, or manually using the *sysfs* interface.

5 LIBRARY DESIGN

The main goal of this paper is to propose an easily extendable design of a network configuration library, with regards to several requirements such as: *object–oriented* design to make the configuration as concise as possible, *extendability* of supported features, *portability* to multiple operating systems, support for several *different means of access* to the configuration interface (for example remote procedure calls) and support for *notifications* of configuration changes.

5.1 PROPOSED SOLUTION

The API and the library implementation is split into two parts. The first being a small statically linked object (`class LibNCFG`) that provides the access API of the library. The constructor of the LibNCFG class detects the capabilities of the underlying operating system determining which functionality will be available to the user later. This solves the technical problem of portability between different kernel space to user space interfaces of different operating systems. Furthermore the class instatiates a *factory* design pattern and handles the construction of specific objects instead of the user. For this it provides several `get_device()` methods that can return different references to a network device object based on the level of abstraction that the user specified. The creation of software devices is handled by `create_soft_device()` methods in a similar way. The same goes for methods that access tables (firewall, routing, etc.). Finally it defines the `define_callback()` and `check_updates()` methods that provide a mechanism for implementing synchronous checks for configuration changes.

The second part of the library and its API is defined in two hierarchies of virtual classes, one for *network devices* and the other one for *tables* (firewalls or routing). This makes it easy to have different implementations of a certain class for different operating systems while preserving the API. These concrete classes are compiled into shared library objects that are dynamically loaded when the LibNCFG object is constructed and the feature is supported by the operating system. The *network device* class hierarchy starts with the most general `class NetDevice` that can access the devices basic information and IP configuration (`{get, set, del}_ip_address()` methods), followed by classes that define more specific methods, for example `{get, set}_hw_address()` for *Ethernet* devices, or `{get, add, del}_slave()` for *link aggregation* devices.

To provide a highly *extendable user space access,* the library is split into a core library, described above, and a series of wrapper applications and libraries that add support for different languages and interface types. This means that support for new interfaces will not require changes in the core library. In addition, support for asynchronous *notifications* can be handled by these wrappers, since it might be impossible for the core library in some cases.

6 CONCLUSION

The goal of this paper was to study and analyse network configuration on Linux and BSD based operating systems, from different points of view and design a library providing a user space interface. The chosen design of the library followed the requirements on: extendability of supported features, supported user space interfaces, operating system portability and support for notifications.

REFERENCES

[1] Robert Love. *Linux Kernel Development, Third edition.* Addison-Wesley, 2012. ISBN 0-672-32946-8.

[2] {Website}. Linux kernel documentation [online]. `https://www.kernel.org/doc/Documentation/`, [cit. 2014-1-2].

[3] {Website}. FreeBSD Handbook [online]. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/`, [cit. 2014-1-2].