Efficient Implementation of Dynamic Protocol Stacks in Linux

Ariane Keller
ETH Zurich, Switzerland
ariane.keller@tik.ee.ethz.ch

Daniel Borkmann
ETH Zurich, Switzerland
HTWK Leipzig, Germany
dborkma@tik.ee.ethz.ch

Wolfgang Mühlbauer
ETH Zurich, Switzerland
muehlbauer@tik.ee.ethz.ch

ABSTRACT

TODO: rewrite abstract - beginning is copied from an old paper... Future network architectures aim at solving the shortcomings of the traditional, static Internet architecture. In order to provide optimal service they have to adapt their functionality to different networking situations. This can be achieved by dividing the networking functionality into modular blocks and combining them as required at runtime. In this paper we address the performance aspect of such architectures and we show that their performance is comparable with the performance of a standard Linux protocol stack.

1. INTRODUCTION

The fast speed of the growth of the Internet and the huge effect on everyday life could lead to the thought that it is perfectly designed and nothing should be changed on the underlying architecture. However, researcher are working continuously on new protocols that improve the communication performance for different communication scenarios. Be it in the area of routing, transport or completely new network architectures. The evaluation of such protocols has shown to be difficult as simulations are not realistic enough and as it is difficult to change anything in standard operating systems protocol stacks. To overcome this difficulty some environments dedicated for testing were built, for example Click [3] and openflow [5] in the routing area or NetFPGA [4] for the evaluation of hardware implementations.

None of these platforms is specifically designed for evaluating protocols on the end nodes and none of these platforms are designed for an adaptation of the protocol stack at run time.

In this paper we present the architecture of a framework that is designed for the following three goals.

1. Provide a platform in which it is easy to test new protocols on end nodes. In order to simplify testing further it should not require any specialized hardware.

2. Provide a platform that imposes as little overhead as possible. This is required that the evaluation of new protocols delivers meaningful results.

3. Provide a platform which allows us to continuously optimize the protocol stack. This is especially useful in mobile scenarios where network characteristics such as delay or packet loss change frequently. Being able to use a protocol optimized for the given network characteristics might improve the connection quality drastically.

Our architecture is based on the ideas of ANA (Autonomic Network Architecture) [2]. In ANA networking functionality is divided into functional blocks that can be arranged as required. ANA does not impose any protocols to be used other than Ethernet rather a path through several functional blocks is built dynamically whenever an application is started. While there exists an implementation of ANA it was only designed as a proof of concept architecture and is not useful for the performance evaluation of actual protocols. Whereas in our work achieving a good performance was one of the main driving goals.

2. LIGHTWEIGHT AUTONOMIC NETWORK ARCHITECTURE (LANA)

LANA provides a framework for setting up protocol stacks not known by todays standard operating systems. Within LANA it is also possible to change the protocol stack at runtime, without communication teardown or application support. These properties build the basis for a networking system that provides protocol stacks that are better targeted to a networking situation than the well known TCP/IP protocol stack.

Similar to the protocol stack of Linux' networking subsystem the protocol stack of LANA is implemented in kernel space and applications can access it over the BSD socket interface. On the other hand, hardware and device drivers are hidden between a virtual link interface. This interface allows on the one side to have multiple virtual interfaces or networks similar to VLANs on different underlying networking technoligies such as Ethernet, Bluetooth or InfiniBand and on the other side ingress and egress points to LANA functional blocks with its packet processing engine. Additionally, virtual link interface devices are represented as usual kernel networking devices and can be managed with standard tools such as `ifconfig`, `ifpps` or `ethtool`.

The setup of the control flow between functional blocks is done with event messages by the Linux notification chain framework. They are used for setup and teardown of data

ANCS 2011 Brooklyn, New York, USA

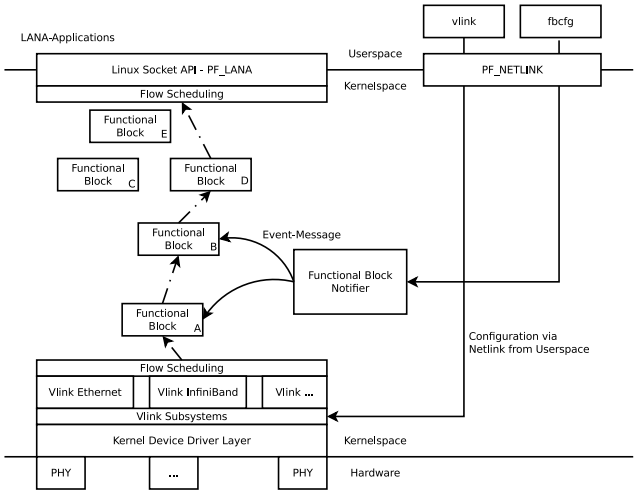

Figure 1: Lana architecture

flow paths between functional blocks or for exchange of other internal functional block data.

Data is transmitted between the functional blocks by function calls and is therefore not being copied between functional blocks. A network packet is processed by the LANAs packet processing engine, which calls receive handlers of functional blocks that are bound to each other in the processing path. Data is then either forwarded to the subsequent functional block or discarded. A certain path can be traversed in ingress or egress direction, thus functional blocks can also flip the packets direction within the packet processing engine. Moreover, the packet processing engine has per-CPU backlog queues so that functional blocks which spawn new network packets can enqueue the like for processing.

There are two special functional blocks - those of the virtual link interface communicating with a network driver and those communicating with BSD sockets. These functional blocks push network packets either in the drivers transmit queue or in the sockets receive queue.

2.1 Configuration Interface

The protocol stack can be configured from user space with the help of a command line tool. The most important commands are summarized below.

- `add`, `rm`: Adds (removes) a functional block from the list of available functional blocks in the kernel.
- `set`: sets properties of a functional block with a `key=value` semantic
- `bind`, `unbind`: Binds (unbinds) a functional block to another in order to be able to send messages to it.
- `replace`: Replaces one functional block with another functional block. The connections between the blocks are maintained. Private data can either be transferred to the new block or dropped.
- `subscribe`, `unsubscribe`: Subscribes (Unsubscribes) one functional block to receive control messages from another functional block.

2.2 Improving the Performance

During the implementation of our framework we have evaluated different possibilities for the integration of our packet processing engine with the Linux kernel. We think the insights gained are interesting for other researcher that have to do fundamental changes on the Linux protocol stack and hence, we summarize them here. Our goal was to be able to process as many *minimum sized Ethernet frames* as the Linux kernel is able to process. In order to compare the performance of the Linux Kernel and the performance of our engine we have dropped all packets in the Linux Kernel protocol stack as soon as they were arrived (TODO: where exactly?). In our system the packets were processed by the fb_eth functional block followed by two fb_dummy functional blocks that were simply forwarding the packets. We can distinguish the following three approaches:

- On each CPU there exists one high priority thread that is responsible for processing LANA packets. This approach leads to a starvation of the interrupt handler (ksoftirqd) and hence the maximal achieved packet rate is only about half as what is achieved by the protocol stack of the Linux kernel. Also changing the priority of the LANA thread to normal only slightly increases the throughput.
- Instead of relying completely on the Scheduler of the Linux Kernel we control preemption and scheduling explicitly. This approach still exhibits scheduling overhead, but it increases the performance to about two thirds of the performance of the Linux Kernel.
- Instead of executing the LANA functions in a dedicated thread they are executed directly in the ksoftirqd function. With this approach approximately 95% of the performance of the Linux kernel is achieved.

The corresponding numbers are listed in Table 1.

mechanism	performance
dedicated kernel thread (high priority)	700000
dedicated kernel thread (normal priority)	750000
dedicated kernel thread (controlled scheduling)	900000
execution in ksoftirqd	1300000
Linux kernel stack	1380000

Table 1: Performance evaluation (pps) of different approaches to receiving packets in the Linux kernel. The packets are 64 Bytes long. The evaluation was done on a TODO: CPU/RAM/NETWORKCARD/KERNEL

2.3 Software Available

The current sofware is available under the GNU General Public License from [1]. In addition to the framework it also includes four functional blocks: Ethernet, Berkeley Packet Filter, Tee (duplicate a packet), and Forward (an empty Block that just forwards the packets to another block). The framework does not need any patching of the Linux kernel but needs a new, 2.6.X kernel.

3. CONCLUSIONS AND FUTURE WORK

We have shown that it is possible to implement a flexible protocol stack that has a similar performance than the default protocol stack in the Linux kernel. This allows for the easy inclusion of new, still to be developed protocols and for the change of the protocol stack at runtime to include for example compression or encryption as the networking conditions change.

In the short term we will compare the performance of real scenarios implemented in our system with the performance of an implementation in other systems (for example default Linux protocol stack or the Click router). In the midterm we will develop a mechanism that automatically sets up a protocol stack for an Application whereby the Application can specify some characteristics the communication channel should have, but not exactly how this has to be achieved. For example the application could require a "reliable communication channel" and a controller would choose between different protocols that provide reliability (e.g., one for wired communication, one for wireless communication, one for wireless, multi-hop communication). The setup of the protocol stack will have to be negotiated between the source and destination node. The end goal will be to have a networked system that requires less configuration as compared to todays networks and that is able to adapt itself to changing network conditions.

4. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n^o257906.

5. REFERENCES

[1] Lightweight Autonomic Network Architecture for the Linux kernel. `http://repo.or.cz/w/ana-net.git` (Jul 11).

[2] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May. The autonomic network architecture (ANA). *Selected Areas in Communications, IEEE Journal on*, 28(1):4 –14, Jan. 2010.

[3] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[4] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, March 2008.