Efficient Implementation of Dynamic Protocol Stacks in Linux

WM: title is ok, but could be more trendy ... I'll think about it

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

Network programming is widely understood as programming strictly defined socket interfaces. Only some frameworks (e.g., ANA, Click, Active Networking) have made a step towards *real* network programming by decomposing networking functionality into small modular blocks that can be assembled in a flexible manner. In this paper, we tackle the challenge of accommodating 3 partially conflicting objectives: (i) high flexibility for network programmers and network application designers, (ii) re-configuration of the network stack at runtime, and (iii) high packet forwarding rates. First experiences with a prototype implementation suggest little performance overhead compared to the standard Linux protocol stack.

1. INTRODUCTION

Beyond doubt, the Internet has grown out of its infancy. A huge variety of networked applications and a diverse range of protocols are available, ranging from protocols for the communication over fibre, cat5 or over the air to protocols supporting specific applications such as p2p, web or voIP. However, the architecture is not designed to also allow for an easy integration of new protocols between these two layers. We argue that an architecture that would not limit innovation to the outer layers would give the Internet another boost. *AK: should also motivate runtime here.*

Some research with this goal was already done in active networking [6, 3], with the Click modular router [4] or with openflow [5]. However, none of the available implementations fulfils the following three partially conflicting objectives.

1. Simple integration and testing of new protocols on end nodes on all layers of the protocol stack.
2. Runtime reconfiguration of the protocol stack in order to allow for even bigger flexibility.
3. High performance packet forwarding rate.

Therefore we propose another architecture that was designed with those three goals in mind. The architecture is based on the ideas of the *Autonomic Network Architecture (ANA)*. [2]. In ANA network functionality is divided into functional blocks that can be combined as required. Each functional block implements a protocol such as *ip, udp, encryption, content centric routing, etc.* ANA does not impose any protocols to be used other than Ethernet, rather it provides a framework that allows for the flexible composition and recomposition of functional blocks to a protocol stack. The existing implementation of ANA shows the feasibility of such a flexible architecture but it suffers sever performance issues. In this paper we present an architecture that allows for a similar functionality than ANA but we show that this flexibility does not have to come with the cost of reduced performance.

ANCS 2011 Brooklyn, New York, USA

2. LIGHTWEIGHT AUTONOMIC NETWORK ARCHITECTURE (LANA)

WM: quickly say first why you propose another architecture; e.g., Click configuration if an kernel cannot be changed at runtime or not happy with Ana's performance; it should be come clear that we try to accommodate conflicting objectives, see my abstract 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.

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 behind 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 technologies 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

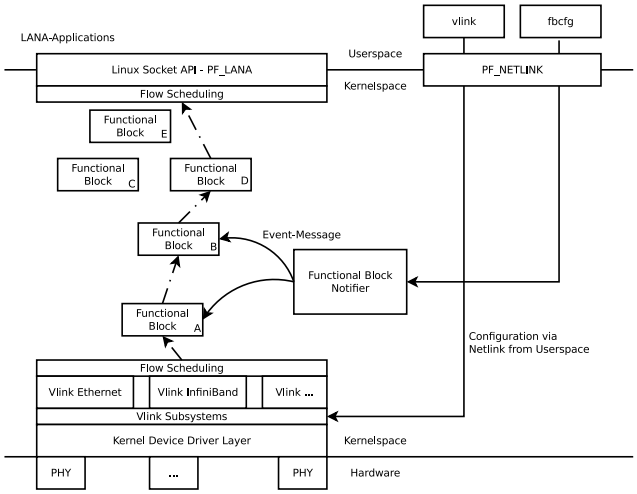

Figure 1: LANA architecture

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 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 interfaces 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 specific 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 event messages from another functional block. (An implicit subscribtion (unsubscribtion) is done on bind (unbind).)

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 researchers 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 bypassed all packets from the Linux Kernel protocol stack into the LANA stack via a `netdev_rx_handler` in bottom half context as soon as they arrived.

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 software 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 (since the ksoftirqd is a low-priority thread).
- Instead of relying completely on the process 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 context. 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)	700.000
Dedicated kernel thread (normal priority)	750.000
Dedicated kernel thread (controlled scheduling)	900.000
Execution in ksoftirqd	1.300.000
Linux kernel networking stack	1.380.000

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 with the kernel packet generator `pktgen` on two directly connected machines with Intel Core 2 Quad Q6600 with 2.40GHz, 4GB RAM, an Intel 82566DC-2 NIC and Linux 3.0rc1.

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), Packet Counter and Forward (an empty Block that just forwards the packets to an-

other block). The framework does not need any patching of the Linux kernel but needs a new 3.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] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela. A survey of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 29(2):7–23, 1999.

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

[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.

[6] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35:80–86, 1997.