Efficient Implementation of Dynamic Protocol Stacks

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 [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. This allows for the experimentation with protocol stacks that are not known by todays standard operating system and it allows for the optimization of the protocol stack at runtime without communication tear down or application support. The existing implementation of ANA shows the feasibility of such a flexible architecture but it suffers sever performance issues. In this paper we present the *Lightweight Autonomic Network Architecture (LANA).* It allows for a similar functionality than ANA but demonstrates that flexibility does not have to come at the cost of reduced performance.

ANCS 2011 Brooklyn, New York, USA

2. LANA

Generally, the LANA network system is built similarly to the network subsystem of the Linux kernel. Applications can send and transmit packets via the BSD socket interface and the actual packet processing is done in a *packet processing engine (PPE)* in the kernel space (figure 1).

The hardware and device drivers interfaces are hidden from the PPE behind a *virtual link interface*, which allows a simple integration of different underlaying networking technologies such as Ethernet, Bluetooth or InfiniBand.

Each functional block is implemented as a Linux kernel module. It offers a *receive handler* that is invoked by the PPE upon packet processing. Functional blocks are built as objects of a specific type. Their build handlers are registered to the LANA core during module registration. Functional blocks can either drop a packet, forward a packet to either ingress or egress direction or duplicate a packet. After having processed the packet it returns the identifier of the next functional block that should process this packet. In addition functional blocks belonging to the virtual link interface will queue the packets in the network drivers transmit queue and functional blocks communicating with BSD sockets will queue the packets in the sockets receive queue.

The PPE is responsible for calling one functional block after the other and for queuing packets that need to be processed.

2.1 Configuration Interface

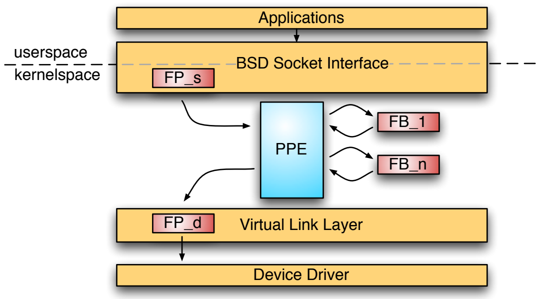

Figure 1: Packet flow in LANA

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.

Within the Linux kernel the notification chain framework is used to propagate those configuration messages to the individual functional blocks.

2.2 Improving the Performance

We have evaluated different possibilities for the integration of the PPE with the Linux kernel. We summarize our insights to provide guidance for researchers that have to do fundamental changes on the Linux protocol stack.

We compared the maximum packet reception rate of the Linux kernel while not doing any packet processing with our architecture. Here packets are forwarded between three functional blocks that do only packet forwarding.

- One high priority LANA thread per CPU achieves approx. half the performance of the default stack. The performance degradation is due to 'starvation' of the software interrupt handler (ksoftirqd). Changing the priority of the LANA thread only slightly increases the throughput (since the ksoftirqd is a low-priority thread).
- Explicit preemption and scheduling control achieves approx. two third of the performance of the default stack. The performance is still reduced by scheduling overhead.
- Execution of the PPE in ksoftirqd context. This approach achieves approximately 95% of the performance of the Linux kernel.

The corresponding numbers are listed in Table 1.

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 (duplication of packets), Packet Counter 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 it requires a new Linux 3.X kernel.

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 in pps with 64 Byte packets. (Intel Core 2 Quad Q6600 with 2.40GHz, 4GB RAM, Intel 82566DC-2 NIC, Linux 3.0rc1)

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. The flexibility allows for the easy inclusion of new, still to be developed protocols and for the change of the protocol stack at runtime. Both might lead to a protocol stack that is better suited for a given networking situation than the well known TCP/IP protocol stack.

In the short-term we will compare the performance of our system with the performance of other systems (e.g., default Linux stack, Click router, etc.). In the mid-term we will work on mechanisms that automatically configures protocol stacks based on the applications as well as the networks needs. 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. `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.