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 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, (ii) re-configuration of the network stack at runtime, and (iii) high packet forwarding rates. First experience with a prototype implementation in Linux suggest little performance overhead compared to the standard Linux protocol stack.

1. INTRODUCTION

Beyond doubt, the Internet has grown out of its infancy and has become a critical infrastructure for private and business applications. Its success is largely due to the plethora of transport media it uses and to the rich set of network applications it offers. Yet, *network programming* is still mainly about programming sockets that form a strictly defined interface between the networking (TCP/IP) and the actual application part (Facebook, VoIP, etc.). What if designers of network applications could even tailor the networking functionality to their needs? We can just speculate about the resulting innovations.

Nowadays, changes in the configuration of a protocol stack usually require applications or even the operating system to be restarted. Such protocol stack changes occur if networking functionality needs to be patched, if the encryption method used is not considered safe anymore or when privacy concerns change. Conceptually, applications should not be harassed by such changes. Therefore, we advocate *run time reconfigurable* protocol stacks. Such protocol stacks could also be used by the various initiatives that work on self* properties in computing to provide an algorithm that configures and adapts the protocol stack autonomously.

Similar objectives were also set by active networking [3], the Click modular router project[4], or OpenFlow [5], etc. Yet, we are not aware of any research that has achieved the following three partially conflicting goals

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 rates.

In this paper, we propose the *Lightweight Autonomic Network Architecture (LANA)*. Our architecture borrows ideas from ANA [2], where network functionality is divided into *functional blocks (FB)* that can be combined as required. Each FB implements a protocol such as *IP, UDP, or content centric routing.* ANA does not impose any protocols to be used , rather it provides a framework that allows for the flexible composition and recomposition of FBs to a protocol stack. This allows for the experimentation with protocol stacks that are not known by todays standard operating systems and it allows for the optimization of protocol stacks 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 contrast to ANA, the proposed LANA architecture is completely written in the Linux kernel and makes use of its optimized structures for packet handling. Additionally, instead of passing packets by copy between individual blocks we use a strict passing by reference approach. Surprisingly, our first experiences with a prototype implementation suggest that we can offer comparable flexibility as ANA but at packet forwarding rates comparable to those of the standard Linux networking stack.

2. LANA: APPROACH

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. The actual packet processing is done in a *packet processing engine (PPE)* in the kernel space. An overview of the architecture is presented in (Figure 1).

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

Each functional block is implemented as a Linux kernel module. Upon module insertion a constructor for the creation of an instance of the FB is registered with the LANA core. Upon configuration of the protocol stack the instances of the FBs are created. The instances register a *receive function* with the PPE. This function is called when a packet needs to be processed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ANCS 2011 Brooklyn, New York, USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

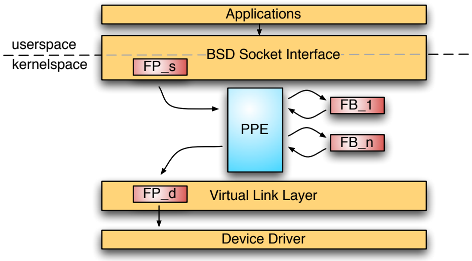

Figure 1: Packet flow in LANA

Functional blocks can either drop a packet, forward a packet to either ingress or egress direction or duplicate a packet. After having processed a packet the FB returns the identifier of the next FB that should process this packet. In addition, FBs belonging to the virtual link interface will queue the packets in the network drivers transmit queue and FBs communicating with BSD sockets will queue the packets in the sockets receive queue.

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

2.1 Implementation

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) an FB from the list of available FBs in the kernel.
- `set`: sets properties of an FB with a `key=value` semantic
- `bind`, `unbind`: Binds (unbinds) an FB to another FB in order to be able to send messages to it.
- `replace`: Replaces one FB with another FB. 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 FBs.

The current software is available under the GNU General Public License from [1]. In addition to the framework it also includes five functional blocks: Ethernet, Berkeley Packet Filter, Tee (duplication of packets), Packet Counter and Forward (an empty block that 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.

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 LANA. In LANA packets are forwarded between three FBs that do only packet forwarding.

- One high priority LANA thread per CPU achieves approx. half the performance of the default Linux 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.
- Explicit preemption and scheduling control achieves approx. two third of the performance of the default stack. The performance degradation is due to scheduling overhead.
- Execution of the PPE in ksoftirqd context. This approach achieves approx. 95% of the performance of default stack.

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 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 configure protocol stacks based on the applications as well as the networks needs. In the long-term a system that requires less configuration as compared to todays networks and that is able to adapt itself to changing network conditions is envisaged.

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.