

Lightweight Built-in Network Monitor in Linux Kernel for self-adaptive IoT Devices

Junyeol LEE*, Taehyun KIM*, Woo-cheol CHO*, Hyuk-jun LEE*

*Department of Computer Science and Engineering, Sogang University, 35 Baekbeom-ro, Mapo-gu, Seoul, Korea

ljy1230@sogang.ac.kr, taebang86@gmail.com, woocheolc@sogang.ac.kr, hyukjunl@sogang.ac.kr

Abstract— Most of the network monitor systems use a user-level network capture library. The user-level library incurs a large overhead and provides inaccurate and insufficient information for self-adaptive networks. For these reasons, we develop a lightweight built-in network monitor running at Linux kernel level for self-adaptive IoT devices.

Keywords—Network monitor, TCP, Packet Latency, Bandwidth, Linux kernel

I. INTRODUCTION

As various IoT devices employing different computing platforms are deployed, it becomes important to provide an OS network stack capable of adapting itself to the availability of hardware resources and user application requirements. For self-adaptation, monitoring the performance of a network stack without affecting the performance of network applications is critical.

However, existing network monitors using a libpcap Linux library show severe limitations. First of all, existing tools using libpcap (e.g. WireShark) incur a large overhead due to moving packet information from kernel space to user space. This consumes both processing power and resources. Second, processing packet information in the user level may lead to producing unreliable monitoring results under a heavy processor load. Third, existing monitoring tools provide only bandwidth information of a network connection and do not give the latency of packets through network stacks. This limitation prevents the Linux network stack from adapting itself according to the latency requirement of a network connection.

To deal with these issues, we propose a lightweight network monitor built into Linux kernel. The proposed method modifies the Linux network stack to monitor packet streams. It efficiently measures and logs both packet latency and bandwidth for network connections with minimal changes to the stack while it does not affect the kernel processing of TCP/IP protocol. In addition, the proposed method extracts essential information needed to provide QoS to different network connections adaptively depending on predetermined requirements. Compared with existing monitoring schemes where incorrect measurements are observed under a heavy load, our method accurately monitors all packet streams and

provides sufficient detail information for self-adaptive control of network stacks.

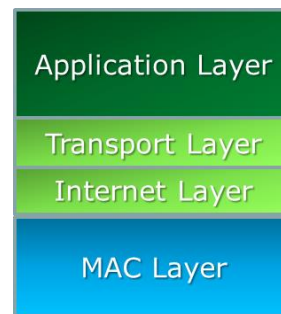
II. BACKGROUND

There are some significant changes in Linux kernel after version 2.6, which can have an impact on the performance of the networking subsystem [1]. In this section we are going to discuss the structure of network stack on Linux kernel and the changes of network system in Linux kernel.

A. NAPI

The most significant change in Linux kernel networking system is NAPI (New API). NAPI is designed to improve the performance of high-speed networking with two main techniques: *interrupt mitigation and packet throttling* [3]. These two techniques reduce CPU load and Linux kernel runs properly under high-traffic environment and while providing remaining CPU resource for other tasks or user applications.

B. Kernel Network Stack



TCP/IP Model

Figure 1. TCP/IP Model Layers

There are four layers in TCP/IP model. Our system monitors three layers below the application layer in kernel space, as shown in Figure 1. An application layer is the part of a user application.

1) MAC Layer: This layer is L2. The latency of this layer depends on MAC header processing performance of NIC and memory performance. Figure 2 shows how packets are copied

from NIC to kernel space. Latency is measured as a packet transition time (i.e. latency) through this layer.

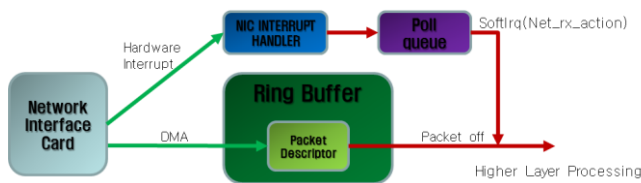


Figure 2. NIC to Kernel Memory Processing Overview

2) Internet Layer: According to the OSI layered model, it is the responsibility of the network layer to perform packet address lookup and routing, and IP is used for the network layer in the TCP/IP protocol suit [4]. We call this layer IP layer in this paper.

3) Transport Layer: This layer is L4. There are many L4 protocols such as TCP, UDP and ICMP, along with several other ones [2]. We consider only TCP protocol in this paper, so we call this layer TCP layer.

III. PROPOSED ARCHITECTURE

In this section, we are going to explain our network monitoring system in detail.

A. Architecture Overview

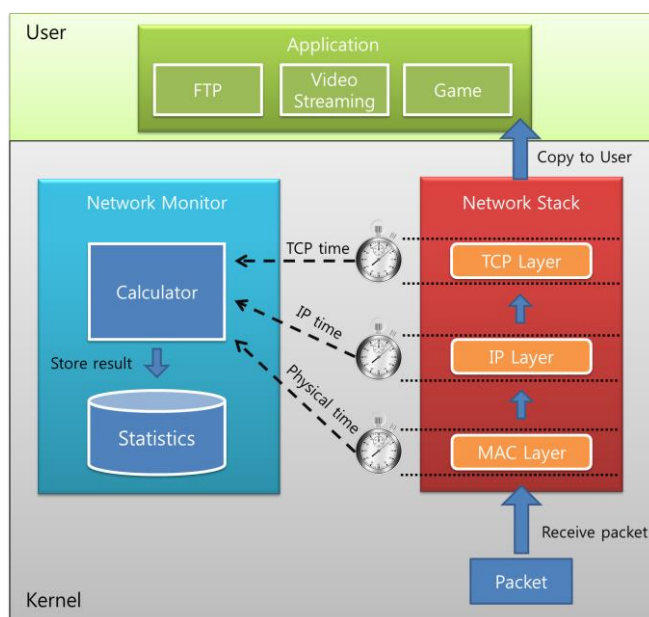


Figure 3. Architecture Overview

Our monitoring system is built in Linux kernel to get special information which cannot be accessed from user-level. Our system is divided into two major parts.

The first part is calculator which collects latency spending on each layer from packets and measures statistics of each connection. Figure 3 describes our architecture overview. The latencies of TCP, IP and physical layer are measured by our modules in each layer inserted in kernel code for TCP/IP

protocol. Adding a module which is not process makes Linux network routine longer, but its overhead is much smaller than lock, context-switch, and IPC (Inter-Process Communication) overhead. Our system frequently accesses kernel structures *socket* and *sk_buff*, also kernel network process accesses and modifies those kernel structures. If monitor module is independent process, monitor process and kernel network process will contend for acquiring lock, and furthermore extra context switching will be occurred. To synchronize monitor process and network process, IPC is essential.

The second part of monitor is a storage which is an optimized structure to store monitoring results efficiently for statistics. Statistics storage is located in *socket* structure. Every connection has own *socket* instance and *socket* instances are managed by Linux kernel. Therefore we do not have to worry about allocating and de-allocating resource. Using kernel data structure which is already managed by Linux kernel guarantees scalability and running properly in SMP (Symmetric Multi-Processing) system.

B. Monitoring Mechanisms

In TCP, the receive side is more complex and abstruse than the send side. Information obtained from the send side is not very attractive from a view of self-adaptive network. In this reason our network monitoring system is monitoring only the receive side of network.

The major goal of our system is monitoring different layers in the network stack, which existing monitoring tools cannot do. Our monitoring system gives transition time through three layers (MAC layer, IP layer, TCP layer) in kernel network stack. In addition, our monitor can provide bandwidth, packet size, and the number of input packets just like existing monitors.

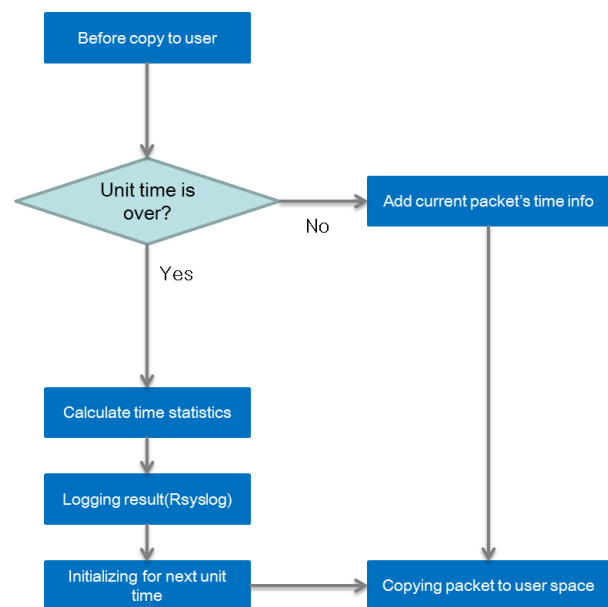


Figure 4. Flow of Monitoring System

To measure average and variance of transition time of each layer, every packet records their own transition time of each

layer. We modify *sk_buff* structure to add some variable for storing time data. *sk_buff* is the most important data structure in the Linux networking code, representing the headers for data that have been received or are about to be transmitted. [2]. Calculator collects meaningful data (transition time of layers, length of packet) from packets before Linux kernel copies data to user buffer and stores that data in statistics storage which is located in *socket* structure. After certain time (logging interval time), calculator reads statistics storage and calculate average and variance of data. When calculation is done, the calculator sends results to the logging system. After sending results, the calculator initializes statistics storage for next results. Figure 4 helps understanding mechanism of this monitor.

C. Logging Method

We use logging system called RSYSLOG (Rocket-fast System for Log Processing). RSYSLOG can deliver over one million messages per second to local destination when limited processing is applied [5]. But this logging system writes data in second storage, it can incur storage bottleneck in gigabit network.

IV. PERFORMANCE MEASUREMENT

We use an in-house TCP benchmark tool to evaluate the performance of the proposed monitoring system. The benchmark tool uses a multi-threaded server-client model. We turn off all offload functions to guarantee fixed packet size.

A. Test Environment

CPU: Intel Core i7-3770 CPU @ 3.40GHz x 4.
OS: Linux kernel 3.10.51 (Ubuntu 12.04).
Memory: 8GB.
Network bandwidth: 100Mbps.

B. Overhead Test

We modified Linux kernel system with our best-efforts to minimize the performance overhead due to our monitoring system, but inevitable performance degradation follows.

A client sends one million 1KB packets to a server and our monitor measures the receive side of a server.

TABLE 1. OVERHEAD OF MONITORING SYSTEM

	Network Throughput (MB/s)	CPU Usage
Pure Kernel	10.78	0.67 %
Modified Kernel	10.78	0.74%

Throughputs of pure kernel and modified kernel are same. CPU utilization of modified kernel is increased about 0.07%. This slight overhead seldom affect other tasks, moreover, most of the CPU overhead is occurred by RSYSLOG. If self-adaptive network stack uses information given by monitor without RSYSLOG, CPU overhead is going to be nearly zero.

C. Capture Accuracy Test

Existing monitoring tools (e.g. WireShark) running in the user space miss quite a few packets under heavy load. We compare our monitor with WireShark which is popular packet sniffer using libcap [6]. A client upload 20MB file to a server. Actual packets are counted at IP layer of send side of kernel.

TABLE 2. PACKET CAPTURE SUCCESS RATE

Tool	Captured Packets/ Actual Packets	Success Rate
Wire Shark	15910/16845	94.44%
Our Monitor	16845/16845	100%

TABLE 2 shows our monitor operates faultless capturing, but WireShark misses about 6% of packets.

V. CONCLUSIONS

This paper explains and evaluates the proposed lightweight built-in network monitor implemented in Linux kernel. Our monitoring system is solving several limitations of existing network monitors. Results of test show that our monitoring system is very accurate and lightweight. Also this monitor provides meaningful information that user level monitors cannot provide. We are going to concentrate on improving our network monitoring system for all protocols, and use this tool for developing self-adaptive network stack for IoT devices.

ACKNOWLEDGMENT

This work was supported by ICT R&D program of MSIP/IITP. [14-824-09-008, The research and development of the self-adaptive software framework for various IoT devices] Hyuk-Jun Lee is a corresponding author.

REFERENCES

- [1] Moore, Andrew, et al. "Architecture of a network monitor." Passive & Active Measurement Workshop. Vol. 2003.
- [2] Benvenuti, Christian. Understanding Linux network internals. " O'Reilly Media, Inc.", 2006.
- [3] Bhattacharya, Shourya P., and Varsha Apte. "A measurement study of the Linux TCP/IP stack performance and scalability on SMP systems." Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on. IEEE, 2006.
- [4] Herbert, Thomas F. The Linux TCP/IP Stack: Networking for Embedded Systems 2E. Charles River Media, 2004.
- [5] RSYSLOG website. [Online]. Available: <http://www.rsyslog.com>
- [6] Qadeer, Mohammed Abdul, et al. "Network traffic analysis and intrusion detection using packet sniffer." Communication Software and Networks, 2010. ICCSN'10. Second International Conference on. IEEE, 2010.



Junyeol Lee received the B.S. degree in Computer Science and Engineering from Sogang University, Seoul, Korea, in 2014. He is currently working toward the MS degree in Computer Science and Engineering at the Sogang University. His research interests include self-adaptive network for IoT, and operating system for embedded system.



Tae-hyun Kim received the B.S. and M.S degree in Computer Science and Engineering from Sogang University, Seoul, Korea, in 2013 and 2015. His research interests include high speed networking, many-core computer architecture, and scheduling algorithm for real-time system. He is a student member of the IEEE.



Woo-cheol Cho received the B.S. and M.S. degrees in Computer Science and Engineering from Sogang University, Seoul, Korea, in 2012 and 2014. He is currently working toward Ph.D. degree in computer science and engineering at the Sogang University. His research interests include shared cache partitioning model for many-core system, architecture for specific big-data application and High speed networking. He is a student member of the IEEE.



Hyuk-Jun Lee received the B.S. degree in Computer Engineering from University of Southern California, Los Angeles, CA in 1993 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1995 and 2001, respectively. From 2001 to 2011, he served as a Senior Engineer in routing technology group at Cisco System, San Jose, CA. He is currently an Associate Professor at the Department of Computer Science and Engineering, Sogang University, Seoul, Korea. His research interests include embedded systems, low-power design, high-speed networking and memory architectures.