GetTCP+: Performance Monitoring System at Transport Layer

Aleksandr A. Sannikov, Olga I. Bogoiavlenskaia, and Iurii A. Bogoiavlenskii

Petrozavodsk State University,
Lenin St., 33, 185640, Petrozavodsk, Russia
`{sannikov,olbgvl,ybgv}@cs.petrsu.ru`
`http://cs.petrsu.ru`

Abstract. Problem of the monitoring of the network performance is important task for different classes of network applications and services. In this paper the system for monitoring of network connections at transport layer is presented. In contrast to existing analogs the monitor is able to provide details on network stack operation visible only at Linux kernel level since the monitor presented operates in both kernel and user space. The paper describes high level architecture of the system, important features of the implementation and testing results.

Keywords: Monitoring, Network Stack, TCP, Linux Kernel.

1 Introduction

Data communication performance monitoring is one of the most important and topical problems since monitoring data provide foundation for network design, development, and administration solutions. The end-to-end path performance plays the key role in this research since it essentially contributes to the end user impression of quality of service available on the path.

In this work we present monitoring system GetTCP+ which collects information on network connections at OSI [1] transport layer and derives performance metrics for the end-to-end paths which are of particular interest for users. The raw data are collected at the OS kernel level which lets monitor to have access to the data unavailable at user's space (e.g. congestion window size) and allows avoiding data distortion and/or variables interpretation problems. Several modules of GetTCP+ are based on facilities of GetTCP monitor [2].

Transport layer and namely Transmission Control Protocol (TCP) [3] is chosen for monitoring since TCP is the only instance that provides flow control solutions and is responsible for distrubuted control of connections access to the network infrastructure. This work does not consider UDP protocol and its extentions, e.g. RTP, since they do not realize flow control algorithms and do not perform data delivery control. Therefore monitoring of TCP behavior reveals essentially wider range of information about end-to-end path performance. Also, for some particular monitoring aspects the developed system captures certain data at network layer, namely Internet Protocol (IP v4 and v6) as well.

Therefore related works in the area presenting several systems of network monitoring which are widely used, e.g. IOS NetFlow [4] and its analogs, *tcpdump* [5], *iperf* [6] and others. Also, one have to mention systems for processing and analysis of monitoring data e.g. Nagios [7], Ganglia [8], *tcptrace* [9]. Distinctly general network monitoring software GetTCP+ collects monitoring data at the OS kernel level and hence gets information that either is totally unavailable at higher OS levels or could not be reliably obtained by network monitoring software of general purpose. Thus, the example demonstrating erroneous estimations of TCP segment size provided by *tcpdump* which was discovered and corrected by the developed system will be presented further.

The rest of the paper is organized as follows. Section 2 describes general system's architecture, identifies modules which use and/or modify GetTCP libraries and presents original facilities of GetTCP+ as well. Section 3 provides some details of the implementation, section 4 contains description of testbed and tests of the system performed. The conclusion offers summary and directions of future research.

2 System Architecture

The system architecture consists of two main units. These are data collection (DC) subsystem and end-to-end path performance metrics storage. The architecture is presented on Fig.1.

Data collection subsystem is based on GetTCP kernel module and *libgettcp* library [2]. The library provides tools for management of control trace points sets and OS kernel module interface for communication and data transfer from kernel space to the user space. Due to the monitoring purposes GetTCP system was significantly modified as well. A new trace point sets for processing connection events and adds segment related events. In particular flow filtering tool, dynamically controlled parameters facility and support of general segmentation offloading mechanism are implemented. Finally, several bugs were fixed as well and the system was ported for modern Linux kernels (v2.6.38-3.1.10). Using *libgettcp* interface and facilities DC subsystem extracts information about end-to-end connection state. Also, it allows processing of single segment-related events.

DC subsystem consists of two parts: kernel module collecting monitoring data and user-space interface transferring data into user space. Filtering mechanism allows extracting flows important for monitoring. When some connection or segment related event rises, trace point handler generates data entry which contains information about the event and the state of network connection. Then this entry is transferred to user-space. At present, GetTCP+ provides following list of end-to-end path metrics: source and destination hosts, maximal congestion window size reached, total volume of transmitted data, number of sent segments, loss rate, maximal segment size, also sequence of congestion window size and round trip time sequence (for each TCP segment transmitted) if required. Thus, with monitoring process organization one could get any information about transport layer connections behavior.

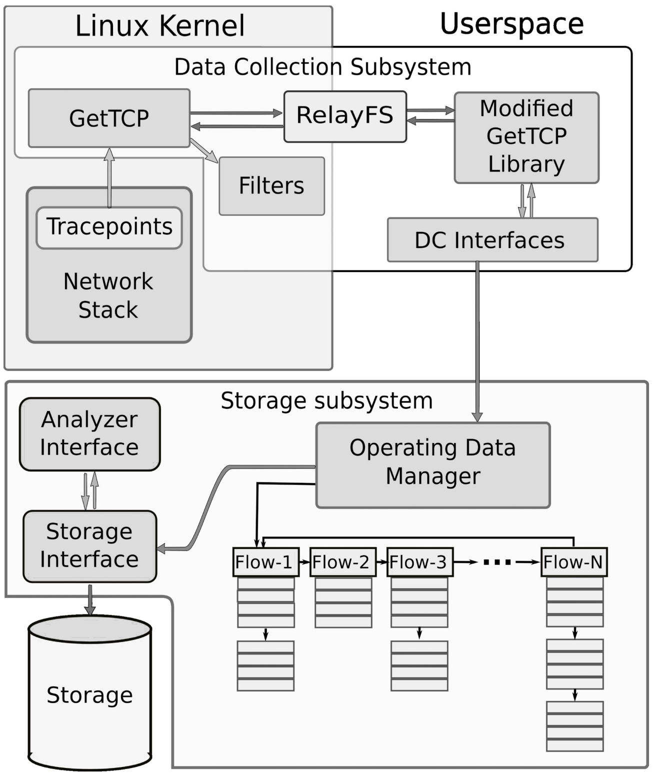

Fig. 1. System architecture

2.1 Storage Subsystem

The storage subsystem consists of three units. These are operating data manager, storage interface and analyzer interface. Operating data manager(ODM) processes running data for on-going TCP flows. A record about every transmitted segment is placed into dynamic memory buffer associated with the flow. When the flow ends, ODM processes the content of the buffer and saves a set of metrics into storage, namely the source and destination, total data sent, flow duration, segment loss rate, MSS, receiver advertised window, mean congestion window, mean RTT. These metrics has been chosen since they are needed to derive current or future TCP performance of the end-to-end path, by direct evaluation or through performance models. After processing ODM instantly removes dynamic buffer to free memory for further usage. On demand ODM can save full sequence of segments data stored in the buffer as well.

The storage interface provides inter-operation between long-time storage and internals of GetTCP+ such as DC and ODM subsystems, hence accumulating the history of end-to-end path performance demonstrated.

The specific features of monitoring data can be denoted as following: data saved never need modification, topicality of the data collected eventually expires, domination of sequential access, data are processed by big slices. Due to these reasons, the storage is based on a file system objects. The information about each sub-network is stored in the separate directory as it is shown on figure 2.

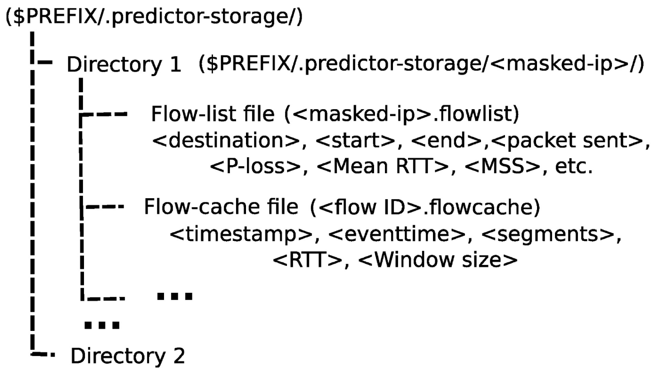
($PREFIX/.predictor-storage/)
Directory 1 ($PREFIX/.predictor-storage/<masked-ip>/)
Flow-list file (<masked-ip>.flowlist)
<destination>, <start>, <end>,<packet sent>,
<P-loss>, <Mean RTT>, <MSS>, etc.
Flow-cache file (<flow ID>.flowcache)
<timestamp>, <eventtime>, <segments>,
<RTT>, <Window size>
...
...
Directory 2

Fig. 2. Storage structure

This directory contains the flow-list file with common information about flows related with specified sub-network. In particular, the information contains: flow ID, host and interface information,total data sent, flow duration and some metrics: loss rate, mean round trip time, mean window size. The example of flow-list records follows:

```
Flow-ID,        Source,    Port,  Dst,            Port,  Start(sec), (usec), End(sec),   (usec), WMax, MSS,  Sent,   Lst, RTT(msec)
1839A6F7310580, 127.0.0.1, 55256, 79.133.201.85, 35091, 1334909306, 151608, 1334909474, 692132, 166,  1424, 147061, 78,  107207.893
18DE58F7310A80, 127.0.0.1, 55512, 79.133.201.85, 35091, 1334909474, 805528, 1334909622, 350665, 166,  1424, 147168, 142, 104185.041
196F2CF7310A80, 127.0.0.1, 55768, 79.133.201.85, 35091, 1334909623, 105733, 1334909756, 216969, 166,  1424, 147020, 43,  103939.189
```

The per-flow cache can be created out of dynamic buffer data records by demand. It contains full sequence of data about segments sent. In the cache file every record contains the time stamp, sequence number, congestion window size, RTT duration etc. Thus, complete description of TCP flow behavior can be reconstructed.

The current implementation of the storage offers three types of data presentation: as a log-file, CSV-formatted file and binary representation. Each of them is available in two modes, namely verbose and standard. In a verbose mode cache files for each flow are stored. Let us notice that verbose mode significantly increases volume of stored data. For example the size of cache-file is equal to 10Mb for 200Mb of transmitted data. At the same time, the size of one record in flow-list file with general information and mean metrics is 200-300 bytes approximately.

First type of data presentation is log-file form. In this case each line for standard mode represents one flow. Such form improves visibility of collected data, but increases volume of data stored. This mode simplifies GetTCP+ deployment. Second type is CSV-formatted file. In this case volume of stored data and its visibility decreases insignificantly, although this mode simplifies automatic processing of monitoring results. Finally, the third form is raw binary data storage. This form is equal to CSV-form by structure and significantly increases performance of the system and reduces volume of data stored. Also some special approaches in data storage can be applicable in this form, e.g. flow indexing.

The interface of analysis subsystem provides access to collected data to any analytical application in necessary form. This subsystem can be implemented as set of plug-in modules according to the requirements of analytical application.

2.2 Tracepoints

Let us consider the set of the trace points implemented more in detail. The set is used to get information about TCP flows and separate segments. The tracepoints define breakpoints inside kernel image and associate handlers with them. When control flow reaches a trace point the correspondent handler invokes. GetTCP+ implements a set of trace point handlers located in the network stack of Linux kernel. For the aims of monitoring four events are essential and henceforth, four tracepoints are defined.

The first two of them are *flow_start_event* and *flow_end_event* which are connected with the start and termination of a TCP flow. The events provide common flow information and their handlers are associated to TCP state machine as it is shown on Figure 3.

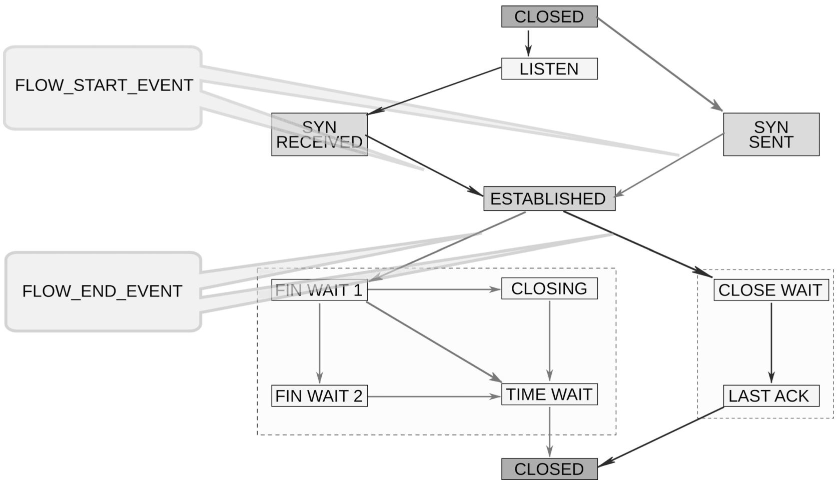

Fig. 3. Flow state-related tracepoints

When the machine changes the state of a connection to "established", *flow_start_event* event is been raised. The event handler performs filtering of the end-to-end path an if it is under monitoring the handler generates record containing information about network device and destination host. When TCP state machine leaves established state *flow_stop_event* is rising. In this case handler provides information about flow duration, maximal congestion window size reached and maximum segment size of the connection.

For per-segment monitoring two other events are used. This events are associated with reached list of unacknowledged segments from retransmission mechanism of TCP, as shown on Figure 4. Each segment sent is added to this list. If the segment is unacknowledged during RTO or treated lost due to SACK information or triple acknowledgment is considered lost and then it is retransmitted by TCP implementation. Thus we can monitor losses. If the segment is acknowledged is removed from the retransmission list. Hence both the segment sent and it's acknowledgment are avaliable the same time. So DC subsystem can obtain full information about the segment.

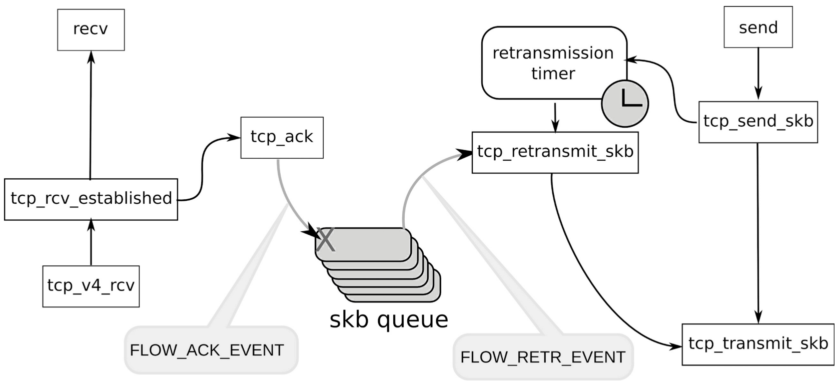

Fig. 4. Segment-related tracepoints

Every unacknowledged segments are considered as lost. The information about connection state is generated by *flow_retr_event* event handler for all such segments.

3 Implementation Special Features

To monitor the flow a special mechanism is required for clustering flows by destination hosts. It should provide the ability of host identification and binding of network flows to such hosts. IP protocol is used for this aims. IP-address can identify both single hosts and sub-networks.

During GetTCP+ development IPv4 and IPv6 facilities were added. It allows collection, storing and processing information about hosts or sub-networks for any IP-based network.

3.1 Segmentation Offloading

The Linux kernel since v2.6.13 uses the set of extensions and improvements of network stack performance. For example, TCP CUBIC flow control algorithm differs (as shown in [10]) from original version proposed in [11]. Some of these changes can affect monitoring tools distorting final results of monitoring. In particular, TSO – TCP segmentation offloading mechanism, may have influenced collected data about TCP flows.

TSO delegates to network interface card the task of splitting big data frames into segments of the size acceptable by data communication network. By this way network stack becomes able to process segments of the sizes several times bigger than MTU. As a result, CPU workload reduces. This technology is especially appropriate in high-performance networks such as 1000BASE-T. As it was shown in [12], TSO significantly increases performance of the network stack. TSO usage looks like transmission of big-size segments at higher layers, because of splitting segments at the network device. So the user space application is not able to estimate the real segment size. Thus, widely used *tcpdump* utility provides erroneous information about segment sizes in high-performance network with the throughput about 50 Mb/sec which enables TSO option. One can observe on Fig. 5 dynamics of segment sizes transmitted during a connection visible by *tcpdump*. At the same time, the real segment size always was equal to 1424 bytes. Meanwhile, according to the data provided by *tcpdump*, the segment size reached 22784 bytes. This is equal at least to 16 transmitted segments.

GetTCP+ prototype is able to operate at kernel's level tracks segmentation offloading and provides correct information about segment size and count. DC subsystem tracks TSO state and parameters passed to it during transmission. One is able to provide correct information about segments characteristics immediately after splitting data to transmit by network interface card under such approach.

3.2 Kernel Module Configuration and Flow Filtering

The run-time configuration interface allows passing specific options to kernel module without reloading, in contrast to the method of passing options as arguments to kernel module. This approach lets an end user change parameters of monitoring dynamically which simplifies the system deployment and exploiting.

At the user space the configuration interfaces are presented through *gettcp_conf_setup(name, value)* function added to *libgettcp* library. This function gets parameter's name and value as a string. Kernel module provides the set of interface functions for secure access to parameter list from handler functions.

The filtering mechanism allows extracting flows important for monitoring. Thus, end user can denote flows related to particular host, sub-network or network interface only. The implementation of filtering in kernel space reduces data flow transmitted to user-space. The filtering is based on two lists: the list of devices and the list of sub-networks. For their management several functions were introduced in the user-space:

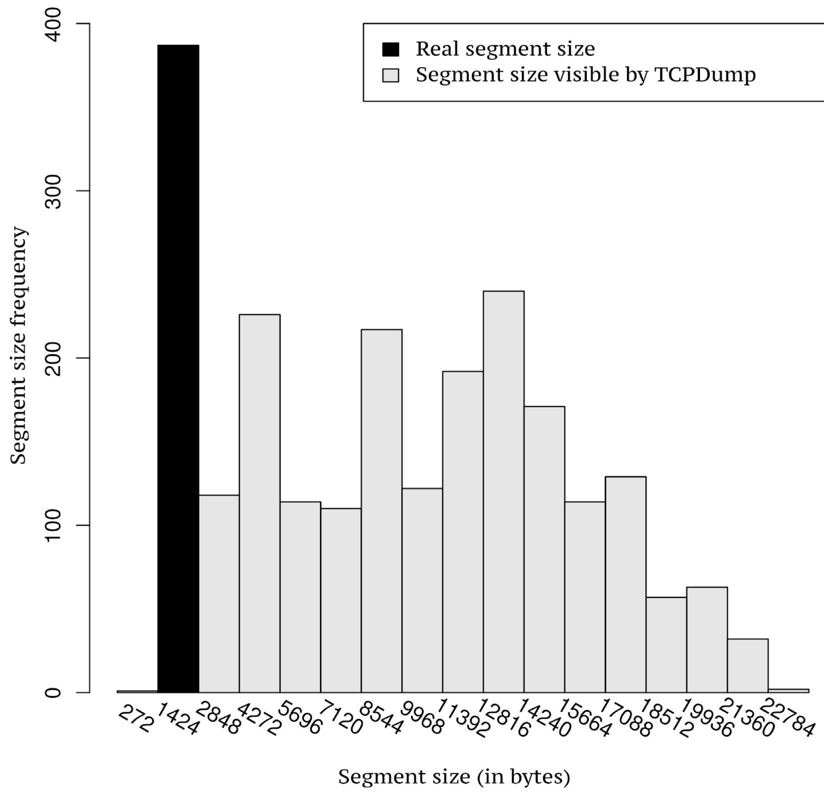

Fig. 5. Histogram of segment sizes provided by tcpdump

- *gettcp_conf_adddev(dname)* – adds device with a specified name to the monitoring list.
  The monitoring is performed only for devices included in this list. Behavior of this filter can be inverted with *DEV_FLTR_EXCLUDED* option.
- *gettcp_conf_addadr(adr, mask)* - adds sub-network or a single host to monitoring list. The filtering is performed by address and network mask. Behavior of this filter is similar to the device filter and can be controlled by *ADDR_FLTR_EXCL.*

The filtering trace point handler is invoked at flow-start. The value of *probed_sock* field from *tcp_sock* structure makes the monitoring for the flow necessary. Further, at the trace points involving processing time, the monitor behavior for the flow is defined by *probed_sock* value. If flow is filtered, all related events are ignored and no data is produced.

4 Testing GetTCP+

GetTCP+ was tested for several network fragments with different characteristics and structure. In particular, the fragments with high performance and the fragments with low throughput, the high round trip time and the loss rate were tested. The four series of experiments were performed from testbed presented on Fig. 6. The monitor GetTCP+ run on the host A which is Intel Celeron 2.20 GHz with RAM 1G and the network connections to Ethernet LAN 1000Base-T and 3G-network on 256Kb/s. The first series of experiments were conducted for the network path with high throughput and low round trip time. This fragment consists of two host connected via router by gigabit Ethernet (A⇒B route on Fig. 6). In this case GetTCP+ was tested under high load.

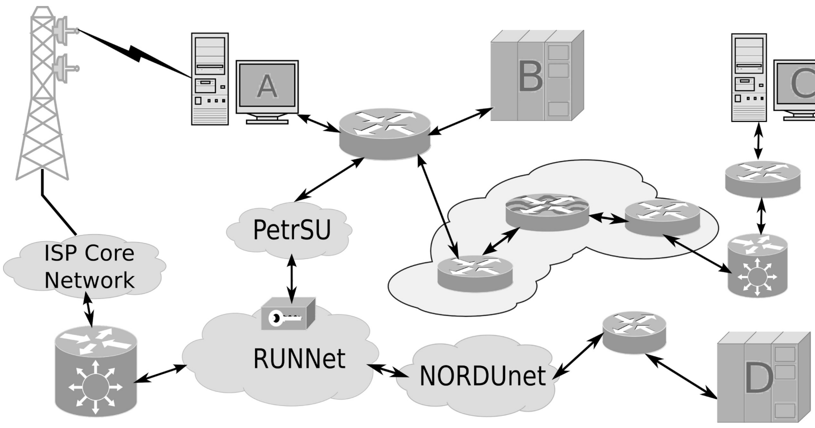

Fig. 6. Testbed

The second and third series of experiments were performed for routes with different throughput and round trip time (i. e. A⇒C and A⇒PetrSU⇒D routes). The fourth series of experiments were performed on the fragment with high loss rate and round trip time (A⇒ISP Core Network⇒D route). During every experiment TCP flow of 200 Mbyte was generated by *iperf* tool at the source host. At the same time these flows were under GetTCP+ monitoring. General information about flows i.e. connection duration, total data sent and average throughput were compared to those provided by *iperf* reports and they correspond completely. The sequences of congestion window sizes provided by cache file follow current TCP NewReno standard and CUBIC implementation for Linux version 2.6.38 completely as well. The round trip time estimations were tested against those provided by *ping* utility and they demonstrated accordance as well.

Also, the delays were estimated. The delays are brought into Linux network stack performance by developed trace point handlers. Linux network stack performance. The execution time of particular functions was measured by *Ftrace* tool. This is internal function's tracer included in the main line Linux kernel since v2.6.27 and it could be used for estimation of the particular function execution time [13]. The average delays measured are following:

Notice, that handlers *tcp_end_event()* and *tcp_start_event()* are invoked once per flow, handler *tcp_retr_event()* is invoked only for lost segments and the loss

Event	Handler	Mean delay
flow_start_event	*tcp_start_event()*	16.904 usec
flow_ack_event	*tcp_ack_event()*	0.628 usec
flow_retr_event	*tcp_retr_event()*	1.172 usec
flow_end_event	*tcp_end_event()*	2.480 usec

rate in the experiments conducted did not exceed 5%. Meanwhile, average processing time is 8.406 usec for standard *tcp_transmit_skb()* function which invokes *tcp_ack_event()* handler, and for standard function *tcp_retransmit_skb()* it is 17.8 usec. Henceforth, GetTCP+ kernel space modules does not bring significant delays into performance of the Linux network stack. Thus, GetTCP+ was tested in different networking environments, and it has shown high stability and performance.

5 Conclusion

The monitoring system GetTCP+ for observation on the end-to-end paths performance at transport layer was developed. This system produces general and/or detailed data of TCP flows performance for the source-destination pairs. The system provides filtering flows, data storage tools, dynamic settings control, using trace points handlers.

In contrast to other existing systems, the certain modules of GetTCP+ operate at OS Linux kernel level. Thus, the system provides accurate and complete information about connection's behavior. It provides correct data which otherwise could be distorted by the monitoring tools operating at user space, e.g. *tcpdump*. The system processes specific important features of network stack, such as TCP segmentation offloading. The interfaces provided by the system allow its integration to the network analysis tools.

For future development we plan to implement an analytical component into the system. Implementation of external interface for storage system will expand the area of GetTCP+ applications as well.

References

1. International Standard ISO/IEC 7498-1, p. 68 (1996)
2. Ponomarev, V.A., Bogoyavlenskaya, O.Y., Bogoyavlenskiy, Y.A.: Configurable Kernel-Level Monitoring System of the TCP Behavior. In: Information Technologies 2010, vol. 1, pp. 54–56 (2010)
3. Allman, M., Paxson, V., Blanton, E.: RFC 5681: TCP Congestion Control (2009), `http://datatracker.ietf.org/doc/rfc5681/`
4. Cisco IOS NetFlow, `http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html`
5. tcpdump/Libpcap public repository, `http://www.tcpdump.org/`
6. Iperf - TCP/UDP Bandwidth Measurement tool, `http://iperf.fr/`

7. Josephsen, D.: Building a Monitoring Infrastructure with Nagios, 1st edn. (2007) ISBN 0-13-223693-1
8. Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing 30 (2004)
9. Tcptrace Homepage, http://www.tcptrace.org/
10. Leith, D.J., Shorten, R.N., McCullagh, G.: Experimental evaluation of Cubic-TCP. In: Proceedings of the 6th International Workshop on Protocols for Fast Long-Distance Networks, March 5-7 (2008)
11. Ha, S., Rhee, I., Xu, L.: CUBIC: A New TCP-Friendly Hight-Speed TCP Variant. ACM SIGOPS Operating Systems Review - Research and Developments in the Linux Kernel 42(5), 64–74 (2008)
12. Linux: TCP Segmentation Offload (TSO), http://kerneltrap.org/node/397
13. Bird, T.: Measuring Function Duration with Ftrace. In: Proceedings of the Linux Symposium, pp. 47–54 (July 2009)