The Klogger Networking Schema

Tal Ben-Nun Yoav Etsion Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
talbn@cs.huji.ac.il

Abstract

Klogger [3] is a fined-grained kernel logging tool, extensible through the ability to define new logging schemata for different kernel subsystems. This paper documents the design and implementation of Klogger's network subsystem schema, whose goal is to track the Linux kernel's (v2.6.22) networking subsystem's activities.

The network schema currently supports IPv4 domain, and specifically the TCP, UDP, ICMP and IP protocols.

1 Klogger's Networking Events

This section describes network event supported by Klogger, helping analysts to better understand how the schema should be used.

In the following subsections, every log level (further documentation in *The Klogger How-To* [2]) contains descriptions of the most common events and their positions.

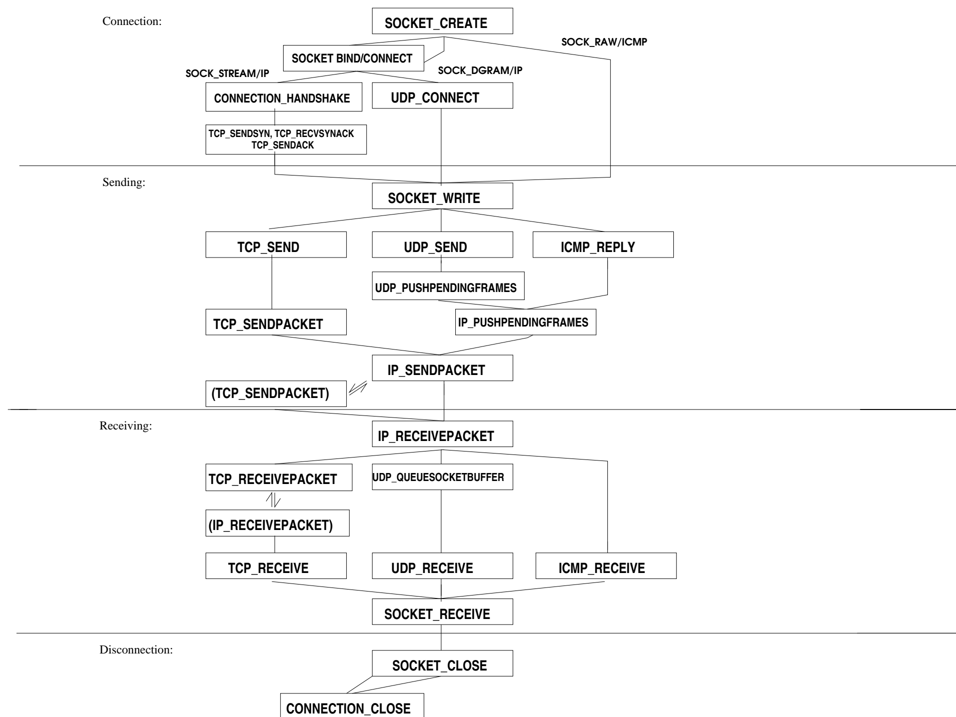

Figure 1: This is a flowchart describing the networking schema's event positions in standard socket procedures.

1.1 Socket/High Level

The high log level focuses on the Linux kernel's main system calls, with events such as *SOCKET_CREATE*, *SOCKET_CLOSE*, *SOCKET_WRITE* and *SOCKET_RECEIVE*.

Socket creation is logged when the newly created socket is assigned to the INET_IPv4 domain (ipv4/af_inet.c). Whenever a socket is called to be closed, the socket closing event is called as well. More simultaneous events in addition to closing include accepting sockets, assigning a socket to be a "listener", assigning an address to a socket and as soon as data is requested to be sent through a socket. When data is received to the socket, though, it is only logged at the very end of the data arrival procedure, so that the analyst will know when the data is outputted to an application, effectively leaving the kernel.

1.2 Connection Level

The connection log level somewhat extends the high log level by adding more useful information and omitting the high log level assign address event to replace it with two different events, *SOCKET_BIND* and *SOCKET_CONNECT*. The *CONNECTION_HANDSHAKE* log event is called whenever a protocol initiates a handshake connection of sorts, such as TCP's handshake mechanism. The *CONNECTION_CLOSE* event is only called after the connection has been closed.

1.3 Protocol Level

The protocol log level contains an extensive list of events which log almost all of the kernel's activities during the protocol's packet processing. Supported events include:

- TCP_SEND[SYN,ACK,SYNACK]
- TCP_RECV[SYN,ACK,SYNACK,URGENTDATA]
- [TCP,UDP,ICMP]_SEND
- [TCP,UDP,ICMP]_RECEIVE
- TCP_SENDPACKET
- TCP_RECEIVEPACKET
- ICMP_REPLY
- [UDP,ICMP]_PUSHPENDINGFRAMES
- UDP_FLUSHPENDINGFRAMES
- UDP_QUEUESOCKETBUFFER

The *PUSHPENDINGFRAMES* event refers to the point where the processed protocol frames/packets are ready to move to the lower level, which is the IP protocol, in our case. The pending frames are pushed to a socket buffer, which will later be used by the IP protocol to send the frames.
The *FLUSHPENDINGFRAMES*, on the other hand, is called whenever there is a "cork" (namely, a hindrance) and all of the pending frames are flushed immediately.
As with the socket level, the *SEND* events are logged the moment they are called and the *RECEIVE* events are logged at the very end of the receiving function. The *TCP_SEND/RECV[SYN,ACK,SYNACK]* events are logged when a given packet is received, processed and identified as a SYN/ACK/SYN-ACK packet upon arrival or whenever a send *[SYN,ACK,SYNACK]SYN/ACK/SYN-ACK* function is invoked.

The *TCP_[SEND,RECEIVE]PACKET* is part of TCP's sending/receiving procedure and, if applicable, the fragmentation/defragmentation procedure. Both of these events are positioned in I/O vector processing loops, because of the intrusive implementation of the TCP protocol (more details can be found in Section 2). The *ICMP_SEND* and *ICMP_REPLY* events are called upon identification of an ICMP packet and contains the type and code of the packets (for example, the ECHO and ECHO_REPLY codes, most commonly used in the "ping" tool).

1.4 IP Level

The IP log level is the closest log level to the network interface card drivers and is considered to be the "lowest" and the most dense (event count wise) of this schema's log levels. It only contains the basic events — *IP_PUSHPENDINGFRAMES*, *IP_FLUSHPENDINGFRAMES*, *IP_SENDPACKET* and *IP_RECEIVEPACKET*.

All these events are simultaneous to the calls from the protocol layer with the exception of *IP_RECEIVEPACKET*, which is called during the accepting and defragmentation of the packets, and IP_SENDPACKET, which is simultaneous unless the IP protocol decides to fragment it. (if so, the event is called after each fragment is complete and is about to be sent)

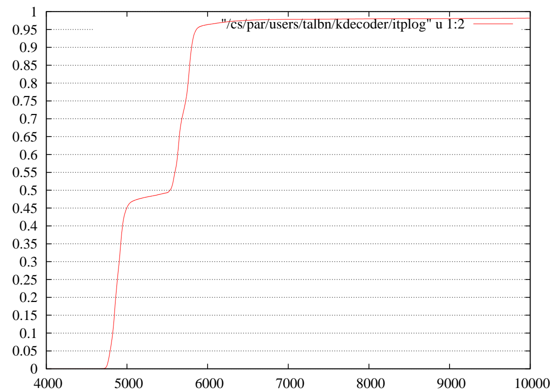

Figure 2: This is a graph showing the bimodal distribution of the IP packets' TCP protocol receive latencies. There are three concentrations: ~5,000 cycles, ~6,000 cycles, and a small amount (approximately 5 percent) of packets which passed the 10,000 cycles. Packet size is a constant 55 bytes.

2 Technical Issues

Using Klogger's network schema, we have analyze the Linux kernel's (v2.6.22) behavior under several known workloads, such as *Netperf* [1], along with various synthetic applications.

The remainder of this section describes some examples of unexpected kernel behavior uncovered by Klogger.

2.1 TCP Packet Latency Bimodal Distribution

While benchmarking the host programs, the Socket Tracker Perl script (included in the Klogger package) logged the average packet transition latency between the IP and TCP layers. This measurement revealed a discrepancy between the distribution's average and median values, which motivated further investigation.

Figure 2 describes the cumulative distribution function (CDF) for that measurement, uncovering a bimodal distribution, centered around 5,000 and 6,000 cycles.

This could be the result of the packet's further processing (or going through a "slow-path", using the Linux kernel's terminology [4, 5]), or header processing if the protocol contained some information on the IP header. One more group of packets (about 5 percent) took more than 10,000 CPU cycles to reach the TCP layer. This behaviour could have been negligible if it was not consistent in all tests.

2.2 TCP's Intrusive Implementation

Operating system textbooks describe the 7-layer OSI networking model as the basis for networking stack design. The three layers that are being discussed in this section are the *Session Layer* (namely, the socket system), the *Transport* layer (protocols such as TCP/UDP/ICMP/IGMP/SCTP) and the *Network* layer (which is used by the IP protocol).

In theory, there should be complete separation between the layers. The Linux kernel's networking subsystem implementation, few of the protocols follow the OSI model almost perfectly. For example, the UDP implementation protocol, which is logged by Klogger, demonstrate a clear layer separation, with small negligible scheduling events occurring from time to time in between the events. On the other hand, the TCP implementation breaks the layer model by merging parts of both Session and Network layers.

There is a plethora of occurrences in which TCP merges with either the IP protocol or the Linux kernel's socket system, from which only some examples, found with Klogger's Networking Schema, will be shown here. The most confusing consequence of TCP's merging in other layers is that the correct order of events (which is the order based on the OSI model and appears in the other protocols), is interfered when sending and receiving data, thus skipping layers.

For example, when sending a large packet, TCP fragments it itself — a task that should be performed in principle by the IP layer — then pushes the fragments to the IP layer. Thus, Klogger's output skip from the TCP_SENDPACKET back and forth to the IP_SENDPACKET event (described in Section 1.4).

Furthermore, to find where TCP fragments/defragments its packets, one would usually look for a function called *tcp_fragment(...)*. A function carrying that name is found in one of the TCP C files, and is even well documented. But even though this function exists, it is practically never called: as an optimization, the actual TCP fragmentation is done while sending the whole data, implemented in an I/O vector clearing loop, without pushing any frames or notifying IP that the packet is a fragment, thus eliminating the receiver's IP protocol's wait for the other packets.

The defragmentation process is also embedded in the data receiving code, as the IP protocol cannot determine whether there are more fragments.

The last intrusive activity which is described in this report is the excessive amount of ACK packets being received after filling a packet window in the receiving machine. Instead of sending an ACK packet back to the sender after a chunk of frames has been processed, an ACK is sent for each freed frame, creating network clutter due to the large amount of ACK packets sent over a short time (the faster the receiving host, the larger the density of ACK packets).

Most of these confusions are likely caused by optimizations and remaining historical code — thus obfuscating the code and making it difficult to understand and maintain. The average programmer trying to augment the Linux kernel's networking subsystem might not understand why the augmented code is not working as expected. We hope Klogger's networking schema will also help to these problems by annotating the existing kernel code with the locations where protocol events take place.

References

[1] Netperf. http://www.netperf.org.

[2] Y. Etsion, T. Ben-Nun, D. Tsafrir, and D. G. Feitelson. *The Klogger HOWTO*. School of Computer Science and Engineering, Hebrew University, Sep 2007.

[3] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with klogger: Experience and insights. In *2nd ACM EuroSys*, pages 259–272, Mar 2007.

[4] T. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems*. Charles River Media, May 2004.

[5] R. Love. *Linux Kernel Development*. Novell Press, Indianapolis, IN, USA, 2nd edition, 2005.