Adventures in TCP/IP

TCP Black Box Logging

BY RANDALL STEWART AND MICHAEL TÜXEN

Evolution of TCP Logging in FreeBSD

4.2 BSD was released in 1983 and included the first TCP implementation in BSD. This version also added support for a facility to debug the TCP implementation. The kernel part, controlled by the kernel option `TCP_DEBUG` (disabled by default), provides a global ring buffer of `TCP_NDEBUG` (default 100) elements and routines to add an entry to the ring buffer whenever a TCP segment is sent or received, a TCP timer expires, or a TCP related protocol user request is processed. These events are only added for sockets, for which the `SOL_SOCKET`-level socket option `SO_DEBUG` was enabled. 4.2 BSD also provided the command line utility `trpt` (transliterate protocol trace), which can read the ring buffer from a live system or core file and print it. It not only prints the TCP header of the sent and received TCP segments, but also the most important parameters of the TCP endpoint when TCP segments are sent or received, TCP timers expire or a TCP related protocol user request is processed. It is important to note, that in case of a panic, the contents of the ring buffer might provide enough information to figure out why the system ended up in the bad state. However, since this facility does not match today's usage of TCP anymore, it was removed in FreeBSD 14. In earlier versions of FreeBSD, building a kernel with a non-default configuration was required.

4.2 BSD was released in 1983 and included the first TCP implementation in BSD.

In 2010, the `siftr` (statistical information for TCP research) kernel module was added to FreeBSD. No changes to the FreeBSD kernel are required, just loading the module to use it. `siftr` is only controlled via `sysctl`-variables. When enabled, controlled by the `sysctl`-variable `net.inet.siftr.enabled`, `siftr` writes its output to a file, controlled by the `sysctl` variable `net.inet.siftr.logfile` (default `/var/log/siftr.log`). The entries, except for the first and last, correspond to a sent or received TCP segment and provide information about the direction, IP addresses and TCP port numbers and internal TCP state. Since it is envisioned to be used in combination with a packet capturing tool like `tcpdump`, no additional information about the TCP segments (for example the TCP header) is stored. Every *n*-th TCP segment will be logged for each TCP connection, seperately for the sent and receive direction. n is controlled by the `sysctl`-variable `net.inet.siftr.ppl`. A TCP port filter controlled by the `sysctl-variable net.inet.siftr.port_filter` can be applied to focus on specific TCP connections. All information is stored in ASCII, therefore no additional userland tool is required to access the information. In the default configuration, only TCP/IPv4 is supported. Adding support for TCP/IPv6 requires a re-compilation of the `siftr` kernel module.

In 2015, a facility was added to the kernel, which is controlled by the kernel option `TCP_PCAP` (disabled by default). If enabled on a non-default kernel, each TCP endpoint contains two ring buffers: one for sent and one for received TCP segments. It should be noted that no additional information, not even the time when a TCP segment was sent or received, is stored. The maximum number of TCP segments in each ring buffer is controlled by the `IPPROTO_TCP`-level socket options `TCP_PCAP_OUT` and `TCP_PCAP_IN`. The default value is controlled by the `sysctl`-variable `net.inet.tcp.tcp_pcap_packets`. Since there is no userland utility to extract the contents of the ring buffers, the use of this feature is limited to analyzing core files. It should be noted that also the TCP payload is logged, which might make it hard to share core files containing such information due to privacy aspects. Support of this facility is planned to be removed in the upcoming version FreeBSD 15.

The latest TCP logging facility, the TCP BBLog (TCP black box logging) was added in 2018. It was initially called TCP BBR (black box recorder), but to avoid confusion with the TCP congestion control called BBR (bottleneck bandwidth and round trip propagation time), it is now called BBLog. BBLog is enabled on all 64-bit platforms of all production releases of FreeBSD. It combines the advantages of `TCP_DEBUG` and `TCP_PCAP` without their disadvantages. Therefore, it is intended to replace both of them. BBLog can be controlled via the `sysctl`-interface and the socket API as described later in this column.

BBLog is enabled on all 64-bit platforms of all production releases of FreeBSD.

Introduction to BBLog

BBLog is controlled by the kernel option `TCP_BLACKBOX` (enabled by default on all 64-bit platforms) and the kernel source code is in `sys/netinet/tcp_log_buf.c` and its corresponding header file `sys/netinet/tcp_log_buf.h`. On a BBLog enabled kernel, there is a device (`/dev/tcp_log`) for providing BBLog information to userland tools, and each TCP endpoint contains a list of BBLog events.

Each event contains a standard set of important TCP state information as well as (optionally) a block of event-specific data. These events are collected to a set limit and when the limit is reached these events may be sent over to a `/dev/tcp_log` which, if open, relays the information to the reading process(s) for recording. Note that if no process has the device open then the data is discarded.

`tcplog_dumper`, from the FreeBSD ports collection, can be used to read from /`dev/tcp_log` as described below.

All FreeBSD TCP stacks have been instrumented with a minimum of the following event types:

- `TCP_LOG_IN` — Generated when a TCP segment arrives.
- `TCP_LOG_OUT` — Generated when a TCP segment is sent.
- `TCP_RTO` — Generated when a timer expires.
- `TCP_LOG_PRU` — Generated when a PRU event is called into the stack.

The TCP RACK and BBR stack generate many other logs; there are currently 72 event types defined in `netinet/tcp_log_buf.h`. These logs instrument a wide variety of conditions and both the TCP BBR and RACK stack even have a verbose mode that can be used when debugging the stack. These verbose options are set through stack specific `sysctl`-variables `net.inet.tcp.rack.misc.verbose` and `net.inet.tcp.bbr.bb_verbose`.

Each TCP endpoint can be in one of the following BBLog states:

- `TCP_LOG_STATE_OFF (0)` — BBLog is disabled.
- `TCP_LOG_STATE_TAIL (1)` — Log only the last events on the connection. Each connection is allotted a finite number (default 5000) of log entries. When the last entry is hit, reuse the first entry overwriting it.
- `TCP_LOG_STATE_HEAD (2)` — Log only the first events processed on the connection up to the limit.
- `TCP_LOG_STATE_HEAD_AUTO (3)` — Log the first events processed on a connection and when you reach the limit dump the data out to the log dumping system for collection.
- `TCP_LOG_STATE_CONTINUAL (4)` — Log all events and when you hit the maximum collected number of events send the data out the log dumping system and start allocating new events.
- `TCP_LOG_STATE_TAIL_AUTO (5)` — Log all events at the tail of a connection and when you hit the limit send the data out to the log dumping system.

Note for general debugging the BBLog state `TCP_LOG_STATE_CONTINUAL` is often used. However in some specific instances (debugging a panic) it is preferable to use the BBLog state `TCP_LOG_STATE_TAIL` such that the last BBLog events are recorded inside the panic dump.

BBLog states can be set when the TCP connection is established or via the socket API. In addition to that, they can be set when a TCP connection fulfills a particular condition. This is called a trace point and they are specified for particular TCP stacks and are identified by a number. One example of a tracepoint is getting `ENOBUF` when the TCP stack calls the IP output routine.

BBLog states can be set when the TCP connection is established or via the socket API.

The contents of each event consists of three parts:

1. A BBLog header containing the IP addresses and TCP port numbers of the TCP connection, the time of the event, an identifier, a reason, and a tag.
2. A set of mandatory state variables of the TCP connection including the TCP connection state and various sequence number variables.
3. A set of optional data like information about send and receive buffer occupancy, TCP header information and further event specific information.

Note that TCP payload information is not contained in any BBLog event, but information about IP addresses and TCP port numbers is included in every BBLog event.

Configuration of BBLog

There are basically two ways of configuring BBLog. The general configuration is done via the `sysctl`-interface and the TCP connection specific configuration is done via the socket API.

Generic Configuration via the `sysctl`-Interface

This is the list of BBog related `sysctl`-variables, which are all under `net.inet.tcp.bb`:

```
[rrs]$ sysctl net.inet.tcp.bb
net.inet.tcp.bb.pcb_ids_tot: 0
net.inet.tcp.bb.pcb_ids_cur: 0
net.inet.tcp.bb.log_auto_all: 1
net.inet.tcp.bb.log_auto_mode: 4
net.inet.tcp.bb.log_auto_ratio: 1
net.inet.tcp.bb.disable_all: 0
net.inet.tcp.bb.log_version: 9
net.inet.tcp.bb.log_id_tcpcb_entries: 0
net.inet.tcp.bb.log_id_tcpcb_limit: 0
net.inet.tcp.bb.log_id_entries: 0
net.inet.tcp.bb.log_id_limit: 0
net.inet.tcp.bb.log_global_entries: 5016
net.inet.tcp.bb.log_global_limit: 5000000
net.inet.tcp.bb.log_session_limit: 5000
net.inet.tcp.bb.log_verbose: 0
net.inet.tcp.bb.tp.count: 0
net.inet.tcp.bb.tp.bbmode: 4
net.inet.tcp.bb.tp.number: 0
```

Using the **sysctl**-interface, BBLog can be enabled for TCP connections. The key **sysctl**-variables for this are **net.inet.tcp.bb.log_auto_all**, **net.inet.tcp.bb.log_auto_mode** and **net.inet.tcp.bb.log_auto_ratio**.

The first **sysctl**-variable to consider is **net.inet.tcp.bb.log_auto_all**. If this variable is set to 1 all connections will be considered for the BBLog ratio. If this value is set to zero, then only connections that have had a **TCP_LOGID** set (see below) will get the BBLog ratio applied to them. In most cases, where the **sysctl**-method is used to enable BBLog, the application probably does not have set a **TCP_LOGID**, so setting **net.inet.tcp.bb.log_auto_all** to 1 assures that every connection will be considered.

The next **sysctl**-variable to set is the **net.inet.tcp.bb.log_auto_ratio**. This value determines 1 in *n* (where *n* is the value provided by setting **net.inet.tcp.bb.log_auto_ratio**) connections will have BBLog enablement applied to them. So, for example, if **net.inet.tcp.bb.log_auto_ratio** is set to 100 then 1 in every 100 connections will have BBLog enabled upon them. If BBLog needs to be enabled for every connection **net.inet.tcp.bb.log_auto_ratio** needs to be set to 1.

The final **sysctl**-variablel to consider is **net.inet.tcp.bb.log_auto_mode**. The value is the numeric constant for the BBLog state. For TCP development, the default could be set to 4 for **TCP_LOG_STATE_CONTINUAL** to log every event that is generated by any connection for debugging purposes.

Some of the other items in the **sysctl**-variable can also be useful, the **net.inet.tcp.bb.log_session_limit** controls how many BBLog events a connection can collect before it has to do something with the data, i.e., either send it off to the collection system or recycle (overwrite) the events. The **net.inet.tcp.bb.log_global_limit** enforces a global system limit on how many total BBLog events the operating system will allow to be allocated.

The last three **sysctl**-variables are related to trace points. **net.inet.tcp.bb.tp.bbmode** specifies the BBLog state to be used if the trace point is triggered. **net.inet.tcp.bb.tp.**

`count` is the number of connections that are allowed to have the specified trace point triggered. For example, if set to 4, then 4 connections can trigger the trace point and after that no others will trigger that specific point (this is to limit the amount of BBLog events generated). `net.inet.tcp.bb.tp.number` specifies the trace point to be enabled.

TCP Connection Specific Configuration via the Socket API

The following `IPPROTO_TCP`-level socket options that can be used to control BBLog on an individual connection:

- `TCP_LOG` — This option sets the BBLog state on a connection. Any use of this socket option overrides any previous setting.
- `TCP_LOGID` — This option is passed a string that when set will be used to name the files generated by the `tcplog_dumper`. It associates the string as an "ID" to be associated with the connection. Note that multiple connections may use the same "ID" string. This is possible because the `tcplog_dumper` also incorporates the IP address and ports in the filename generated.
- `TCP_LOGBUF` — This socket option can be used to read data from the current connections logging buffer. Normally this is not used and instead `/dev/tcp_log` is read from by a general purpose tool such as the `tcplog_dumper` (which reads and stores the BBLogs). But this, as an alternative, allows a user process to collect a number of logs.
- `TCP_LOGDUMP` — This socket option directs the BBLog system to dump any records that are in queue on the connection to `/dev/tcp_log`. If no dump reason or ID has been given then the system default for the type of logging underway is used in any "reason" field inside the dump file.
- `TCP_LOGDUMPID` — This socket option, like `TCP_LOGDUMP`, directs the BBLog system to dump out any records to `/dev/tcp_log`, but in addition it specifies a specific user given "reason" for the output which will be included in the BBlog "reason" field.
- `TCP_LOG_TAG` — This option associates an additional "tag" in the form of a string with all BBLog records for this connection.

For example, if access to the source code of the program using a TCP connection is available, the BBLog state of the connection can be set to `TCP_LOG_STATE_CONTINUAL` using the `TCP_LOG` socket option:

```
#include <netinet/tcp_log_buf.h>

int err;
int log_state = TCP_LOG_STATE_CONTINUAL;
err = setsockopt(sd, IPPROTO_TCP, TCP_LOG, &log_state, sizeof(int));
```

This code can also be used for any other BBLog state mentioned earlier.

If no access to the source code is available, one can use with root privileges

```
tcpsso -i id TCP_LOG 4
```

where `id` is the `inp_gencnt`, which can be determined by running `sockstat -iPtcp. 4` is the numeric value of `TCP_LOG_STATE_CONTINUAL`.

Generating BBLog Files

Before enabling BBLog on a specific TCP connection one needs to first make sure that the collection of BBLogs is taking place. FreeBSD has a tool designed for that called tcplog_dumper which is available in the ports tree (net/tcplog_dumper). It can be installed by running with root privileges:

```
pkg install tcplog_dumper
```

Adding

```
tcplog_dumper_enable="YES"
```

to the file /etc/rc.conf will start the daemon automatically after the next reboot. It can also be started a daemon by manually running with root privileges:

```
tcplog_dumper -d
```

By default tcplog_dumper will collect BBLog's in the directory /var/log/tcplog_dumps. There are several other options which are supported including:

- -J — This option will cause the tcplog_dumper to output compressed files with xz.
- -D directory path — Store the files collected in the directory path specified, not the default. This can also be controlled by the rc.conf variable tcplog_dumper_basedir.

The tcplog_dumper will output pcapng (pcap next generation) files. pcapng supports storing meta information in addition to packet information. For TCP_LOG_IN and TCP_LOG_OUT events, tcplog_dumper generates an IP header from the event (so except for the source and destination IP address, the fields in the IP header might not be as they have been on the wire), uses the TCP header from the event (which means it is as the segment was on the wire) and adds a dummy payload of the correct length. For each TCP connection, tcplog_dumper creates a series of files and will put roughly 5000 BBLog events in each file numbered in sequence .0, .1, .2 etc. The following is an example of a series of 7 files for a single TCP connection::

```
[rrs]$ ls /var/log/tcplog_dumps/
UNKNOWN_18262_10.1.1.1_9999.0.pcapng  UNKNOWN_18262_10.1.1.1_9999.4.pcapng
UNKNOWN_18262_10.1.1.1_9999.1.pcapng  UNKNOWN_18262_10.1.1.1_9999.5.pcapng
UNKNOWN_18262_10.1.1.1_9999.2.pcapng  UNKNOWN_18262_10.1.1.1_9999.6.pcapng
UNKNOWN_18262_10.1.1.1_9999.3.pcapng records
```

So the TCP_LOGID was not set on the connection, one of the TCP ports was 18262, the other TCP port 9999 and the remote IPv4 address is 10.1.1.1.

Generating BBLog files from a core dump is currently being worked on. A debugger will be used to extract the information and provide it to tcplog_dumper for actually writing the BBLog files.

Reading BBLog Files

There are two easily accessible tools that can read BBLog files.These are read_bbrlog and wireshark, both available as ports or packages.

read_bbrlog

read_bbrlog is a small program that will read a series of BBLog files and display each log entry in text form. It needs to be given the prefix of the BBLog files as the input source and it finds all of the files associated with that tcp connection and prints out to stdout each event in text form. Note that there is also an option to redirect the output to a file (highly recommended since lots of data will be displayed). Here is an example on how to run read_bbrlog:

```
[rrs]$ read_bbrlog -i UNKNOWN_18262_10.1.1.1_9999 -o
my_output_file.txt -e Files:7 Processed 30964 records Saw
30964 records from stackid:3 total_missed:0 dups:0
```

In this case three options are used: `-i input` where the input argument is the base connection id, i.e., the text displayed by `ls` minus the `.X.pcapng`. The `-o outfile` to redirect output to the output file `my_output_file.txt` and finally the `-e` option which is typically used to put out "extended" output which is more verbose.

Here is a small clip from the file `my_output_file.txt` to give a flavor of the data presented. Note due to the large line length some of the data for display has been truncated off:

```
106565924 0 rack [50] PKT_OUT    Sent(0) 763046978:5  (PUS|ACK fas:0 bas:1) bw:208.00 bps(26)
                      avail:5 cw:14480 scw:14480 rw:65535 flt:0 (spo:64 ip:0)
106565979 0 rack [55] TCP_HYSTART  -- New round begins round:1 ends:763046983 cwnd:14480
106565982 0 rack [3] BBR_PACING_CALC Old rack burst mitigation len:5 slot:0 trperms:369
106565985 0 rack [3] TIMERSTAR  type:TLP(timer:4) srtt:39001 (rttvar:17063 * 4) rttmin:30000
106565986 0 rack [1] USERSEND   avail:5 pending:5 snd_una:763046978 snd_max:763046983 out:5
106565986 0 rack [0] TCP_LOG_PRU  pru_method:SEND (9) err:0
106607480 0 rack [2] IN         Ack:Normal 5 (PUS|ACK) off:32 out:5 lenin:5 avail:5 cw:14480
                                rw:4000512 una:763046978 ack:763046983
```

This shows that a 5 byte packet was sent at timemark 106565924 and sequence number 763046978. The congestion window at the time was 14480 bytes and the flight size (flt) was 0. No pacing was engaged. About 41 milliseconds (41,494 i.e. 106604046 - 106607480) an acknowledgement was received for those bytes.

Wireshark

`wireshark` and `tshark` can also be used to display BBLog files. They only operate on individual files, not on a file series as `read_bbrlog` does. Currently no event specific information will be displayed. For `TCP_LOG_IN` and `TCP_LOG_OUT` events the BBLog information is shown in the Frame Information. For all other events, the BBLog information is directly shown.

RANDALL STEWART (rrs@freebsd.org) has been an operating system developer for over 40 years and a FreeBSD developer since 2006. He specializes in Transports including TCP and SCTP but has also been known to poke into other areas of the operating system. He is currently an independent consultant.

MICHAEL TÜXEN (tuexen@freebsd.org) is a professor at the Münster University of Applied Sciences, a part-time contractor for Netflix, and a FreeBSD source committer since 2009. His focus is on transport protocols like SCTP and TCP, their standardization at the IETF and their implementation in FreeBSD.