Definitions of TCP/IP Network Connection Metrics

S TERRY BRUGGER
University of California, Davis

January 6, 2007

Abstract

We propose a working definition for what exactly comprises a TCP/IP network connection, and exactly how to extract connection metrics from the viewpoint of a third-party observer. Our definitions should be formal enough to allow two different researchers to obtain the same measurement from the same network trace.

1 Introduction

In order to do analysis based on various metrics of TCP network connections, researchers need to agree on exactly what those metrics are. To first order, we must agree on which packets do and do not belong to a given connection. Given this, we need to agree on exactly how to calculate the connection metrics we're interested in from a given trace. We are interested in this from the viewpoint of a third-party passive observer, as this is how we collect network data without affecting the activity on the network.

This paper presents a definition for TCP connections, and how to calculate TCP connection metrics based on state diagrams, based on a reading of RFC793 from the viewpoint of a third party. The metrics themselves have been used by the network intrusion detection community, and we expect they will be of interest to anyone desiring a deeper understanding of networks. We hope that people adopt these metrics for their research, however we do not believe that these are the end-all-be-all definitions. Instead, we offer these working definitions as a proposal in the hopes of generating discussion on how best to measure TCP connections.

2 Definition of a TCP Connection

We will begin with a very basic definition: A TCP Connection is a set of packets between a pair of sockets (Host IP and port pairs). Since the same pair of sockets may be used for multiple connections, this section will further refine this definition.

In a perfect world, the definition of a TCP connection is easy: It is all the packets starting with the initial SYN packet, up to and including the ACK of the second FIN (Postal 1981). Of course, the RFC acknowledges that we don't live in a perfect world, and it defines how hosts should act in a variety of situations. It is easy to extend the above definition to cover some of these, for example, we can say up to and include in the ACK of the second FIN, or a connection RST packet; but do we count a SYN packet in the middle of the current sequence as part of the current connection, or as a separate connection? Do we count a non-SYN packet that is not part of an ESTABLISHED connection to be part of a connection at all?

To that end, we always require a SYN packet to begin a connection. If we don't see a SYN, and we don't have a record of the connection already, we must assume that the connection was already begun before we started observation, hence we do not have a complete record of the connection, and we will not attempt to record metrics on it. It is true that an attacker may attempt to attack the network stack by sending in a malicious packet (for instance a "Christmas tree packet", which has all of the TCP flags set(Miller 2000)) without having already established a connection, however this is not a connection, and any desire to include it in overall network characterization should be done separately from the connection metrics.

Once we see the initial SYN packet, we include all packets between the source and destination sockets in the metrics for that connection. This includes packets that are invalid for whatever reason; for instance packets with incorrect checksums, or out of sequence packets.

A connection ends when:

- Both sides have sent a FIN, which is acknowledged.
- One side sends a RST.
- A packet goes unacknowledged for more than TIMEOUT seconds, where we use the 300 second TIMEOUT suggested by RFC 793.
- There is no activity on the connection for more than TIMEOUT seconds.

The first two are fairly straight forward. When we combine our definition of when a connection begins with the third, it brings with it an important implication: even if a connection never reaches the ESTABLISHED state on the hosts, we include it as a connection in the connection metrics. This means that if a firewall blocks the connection, we will still record it as a connection consisting of one packet. If the source tries to resend the SYN packet, lets say twice, then we'll record it as a connection with two duplicate packets. The fourth definition is really just a generalization of the third, however we call it out separately as we expect it will generate some discussion. The TCP specification indicates that once a connection is established, it will stay established as long as both hosts keep it established. It may be that during this idle time, one of the hosts is removed from the network (shutdown, unplugged, etc), at which point it will, in essence, terminate the connection, without being able to let the other host know. This condition is known as a half-open connection. Our

problem is that, given our point of observation, we can not ascertain when this has occurred. As such, we must infer it through the lack of any activity over some time period. If one accepts this, then the question becomes how long that timeout should be. We propose using the same value as we use for closing the connection when it is in the TIME-WAIT state.

3 Definitions of TCP Connection Metrics

Given the above definition of a connection, we will here define various metrics that we might use in characterizing connections. We follow the subsection for a Segment Arriving in the event processing model in RFC 793 and note where we count various characteristics. Due to our perspective, there is actually little overlap with the endpoint-model that RFC 793 uses, so we duplicate some information as necessary such that the reader does not require a copy of the RFC as they read though. Some areas we differ from the RFC are: we ignore the security and precedence checks, as those features of TCP/IP are not used on contemporary networks; and we also assume a three-way handshake to establish a connection, as it seems the four-way handshake has never borne out.

3.1 Metrics to collect

An exhaustive list of the connection metrics that we collect as we follow the event processing model is as follows:

- Number of packets
- Duration (time of last packet minus starting time), in seconds
- Number control packets
- Number data packets (Number of packets minus Number control packets)
- Bytes transfered
- Data bytes transfered
- Urgent
- Resend
- Wrong resend
- Duplicate ACK
- Wrong ACK
- Wrong data packet size
- Window exceeded
- Hole
- Connection errors
- Reset connection
- Other errors

- Disconnection errors
- Fragmented packets
- Bad fragment
- SYN-ONLY
- SYN-ACK
- Idle connection
- Half-open connection

All of these metrics – except the last four – should be stored in the connection record, and can be reported when that record is closed. The last four can be reported based on the state of the record when it is closed.

All of these metrics should be reported as counts (as opposed to rates), save for duration, which should be the number of seconds, and may be a real number (clock resolution permitting).

3.2 Connection record

We use a modified form of the TCB presented in RFC 793 Section 3.2. Since we're a third party observer to the connection, we attempt to capture the state at both ends of the connection, rather than the definitive information that either host uses.

Information on the source of the connection (the machine that sent the initial SYN):

SRC.UNA earliest sequence number the source has sent that has not been acknowledged by the dest – if all segments the source has sent have been acknowledged, this will be equal to the last ACK

SRC.NXT the next sequence number the source is going to use

SRC.WND the size of the window on the source machine

SRC.UP pointer to the last urgent sequence that the source machine should receive

SRC.UF flag set to true when the urgent pointer should be examined

SRC.WL1 sequence number used for last window update on source machine

SRC.WL2 sequence number of last source window update that the dest acknowledged

SRC.FIN sequence number of the source's FIN packet

Information on the destination of the connection:

DST.UNA earliest sequence number the dest has sent that has not been acknowledged by the source – if all segments the dest has sent have been acknowledged, this will be equal to the last ACK

DST.NXT the next sequence number the dest is going to use

DST.WND the size of the window on the dest machine

DST.UP pointer to the last urgent sequence that the dest machine should receive

DST.UF flag set to true when the urgent pointer should be examined

DST.WL1 sequence number used for last window update on dest machine

DST.WL2 sequence number of last dest window update that the source acknowledged

DST.FIN sequence number of the dest's FIN packet

When looking at an individual packet, we may consider SND to be whomever sent the packet: either SRC or DST, and RCV to be the other. We use the same definition for SEG given in RFC 793, with the clarification that SEG.LEN is the data length of the segment.

3.3 Segment Arriving

Since we're taking the viewpoint of a third-party observer to the connection, the only event we're interested in is "Segment Arriving". Of course, to us, the segment isn't really arriving, a better term might be "Segment Observed".

A note on sequence comparisons: since sequence numbers run between zero and $2^{32} - 1$, then repeat; RFC 793 notes that all sequence comparisons should be done $mod(2^{32})$, while this is not strictly correct, it is easy enough for TCP implementations to check that $SND.UNA <= SEG.ACK <= SND.NXT$, taking into account the special case when $SND.NXT < SND.UNA$. This is not sufficient when collecting metrics and we want to determine if ACKs are being duplicated, if a segment is being ACKed before it arrives, or any other of a number of scenarios. As such, whenever we note that a less than comparison (including the $<$ portion of $<=$), such as $A < B$ is being done, where either A or B is $SEG.SEQ$ or $SEG.ACK$, then if A is not less than B, check if A is past halfway on the set of possible sequence numbers ($> 2^{31}$), B is below the halfway mark, and the difference between them is at least 2^{31}. In formal notation:

$$A < B \text{ or } (A > 2^{31} \text{ and } B < 2^{31} \text{ and } A - B > 2^{31})$$

This may only be a heuristic, however it should be sufficient to classify a packet as either a bad resend, or out-of-sequence.

3.3.1 Fragmented packets

For everything that follows, we assume that we're working with complete segments, just as RFC 793 does. Hence, when a packet arrives with the "More Fragments" (MF) flag set (that is part of a connection that we either have a record for, or it has the SYN flag set), it should be queued up in a fragmented packets queue. Any packet without the MF flag set and with the fragment offset > 0 will be assumed to be the last fragment. When it arrives, the fragmentation pre-processor should:

- verify that all the fragments were received (there are no holes)

- verify that none of the fragments overlap
- verify that none of the fragments (including the last one) have the "Don't Fragment" (DF) flag set.

If there are any holes, the packet is unusable, so the "Fragmented packets" and "Bad fragment" counters should be incremented, and processing on the packet should stop. If any of the conditionals do not hold, then the "Bad fragment" counter should be incremented. In either case, the "Fragmented packets" counter should be incremented.

Since only the first fragment has the TCP portion of the header, we must store the full header from the first fragment and use it for all the segment information we require. We will need to substitute in the full length of the packet as SEG.LEN. This measure will not account for the extra transmission overhead of the extra headers on all the fragments. This is fair because we are primarily interested in the connection metrics at this point. General network metrics, such as the total number of bytes sent or received on a given link, can be collected independently of the connection. The same argument holds for justification for dropping packets with missing fragments before processing them as part of the connection.

Should the last fragment never arrive, then we may end up counting the packet as a hole, below. If we don't count it as a hole, that will mean that it was resent. Unfortunately, if it is not fragmented when it is resent, we will not know to increase the "Resent" counter. If it is, however, fragmented, then the resend will show up as a bad fragment due to overlap between the original and the resent fragments. When a connection is closed, a count should be made of any segments from that connection still on the fragmentation queue, and that number added to the "Bad fragments" count.

The primary problem with this approach is that if the fragments arrive out of order – in particular if a fragment arrives after the last fragment – it will be recorded as a bad fragment. Considering how infrequently fragmented packets are encountered in practice anymore, we are not too concerned with this.

3.3.2 No Record

If we don't have a record of the connection, it could mean that the connection was established before we started observing the traffic, or that this packet is not part of an established connection. In the later case, we don't know if the connection is in the CLOSED or LISTEN state on the receiving host, so we continue to process this packet.

If the packet is a RST, ignore it.

If the packet is an ACK, then it must be part of a connection that was established before we began listening, so ignore it.

If the packet is a SYN, create a new connection record with

- the two sockets
- the starting time of the connection and time of last packet

- set SRC.UNA to SEG.SEQ
- set SRC.NXT to $SEG.SEQ + 1$
- set SRC.WND to SEG.WND
- set SRC.WL1 to SEG.SEQ
- set SRC.UP and SRC.WL2 to zero
- set SRC.UF to false
- set all the DST fields to zero
- set the state to SYN-RECEIVED
- set the "Number of packets" counter to one
- set the "Bytes transfered" to the total length of the packet
- if the SEG.LEN is zero, set the "Number controller packets" counter to one, otherwise increment the "Data bytes transfered" by SEG.LEN and set the "Wrong data packet size" counter to one
- set all other counters to zero

3.3.3 SYN-RECEIVED

When we're in the SYN-RECEIVED state, we expect to see a SYN-ACK.

For all packets:

- increment the "Number of packets" counter
- increment the "Bytes transfered" by the total length of the packet
- if the SEG.LEN is zero, increment the "Number controller packets" counter, otherwise increment "Data bytes transfered" by SEG.LEN and increment the "Wrong data packet size" counter
- set time of last packet

If the packet is not an ACK:

- increment the "Connection errors" counter
- stop processing this packet

Otherwise, the packet is an ACK, so check the acknowledgment number. If $SEG.ACK! = SRC.NXT$:

- increment the "Connection errors" counter
- stop processing this packet

If the ACK is fine, check if the RST flag is set. If so:

- increment the "Connection errors" counter
- increment the "Reset connection" counter
- close the record

If the SYN flag is set, then this is the SYN-ACK we expect:

- set DST.UNA to SEG.SEQ
- set DST.NXT to $SEG.SEQ + 1$
- set DST.WND to SEG.WND
- set DST.WL1 to SEG.SEQ
- set DST.UP and DST.WL2 to zero
- set DST.UF to false
- set SRC.UNA to SEG.ACK
- set SRC.NXT to SEG.ACK
- set the state to SYN-ACK
- stop processing this packet

If we made it this far, the packet is an ACK, but not a SYN-ACK, so:

- increment the "Connection errors" counter
- stop processing this packet

3.3.4 SYN-ACK and later

The SYN-ACK state is what we call when the source is in the ESTABLISHED state and the destination is in the SYN-RECEIVED state. For this state, and all the ones that follow, we follow some common steps.

For all packets:

- increment the "Number of packets" counter
- increment the "Bytes transfered" by the total length of the packet
- if the SEG.LEN is zero, increment the "Number controller packets" counter, otherwise increment "Data bytes transfered" by SEG.LEN
- set time of last packet

First, check that the sequence number is acceptable.

- If $SEG.SEQ < SND.UNA$, then the sender is resending a packet that has already been acknowledged:
  - increment the "Wrong resend" counter
  - stop processing this packet
- If $SEG.SEQ + SEG.LEN - 1 > SND.UNA + RCV.WND$, then the sender is sending a packet that will exceed the receiver's window:
  - if the state is SYN-ACK, increment the "Connection errors" counter
  - increment the "Window exceeded" counter
  - while an end-host would stop processing the packet at this point, we must consider that we failed to observe a packet or that the end-host will be lenient in what it accepts
- If $SEG.SEQ < SND.NXT$, then this is a resend:

- increment the "Resend" counter
- stop processing this packet

- Now see if the packets are out of order: If $SND.NXT < SEG.SEQ$, then this packet is either arriving after one before it in the sequence, or there is a hole in the sequence:
  - If the sequence of any packet in the other direction on the out-of-sequence queue is $< SEG.ACK$, continue processing (we will process the packets in the other direction below when we examine the ACK field)
  - Otherwise, push it on an "out-of-sequence" queue, which we will process later
- The remaining case is that $SEG.SEQ = SND.NXT$, which is what we want and expect, so we continue to process it.

Second, if the RST flag is set:

- increment the "Reset connection" counter
- if the $SEG.LEN > 0$, increment the "Wrong data packet size" counter
- if the state is "SYN-ACK", increment the "Connection errors" counter
- close the record

Third, if the SYN flag is set:

- if the state is "SYN-ACK", increment the "Connection errors" counter, otherwise set the "Other errors" counter
- if the $SEG.LEN > 0$, increment the "Wrong data packet size" counter
- stop processing this packet

Fourth, check the ACK field:

- If the ACK flag is not set:
  - if the state is "SYN-ACK", increment the "Connection errors" counter, otherwise set the "Other errors" counter
  - stop processing this packet
- SYN-ACK state:
  - If $SEG.ACK != DST.UNA + 1$:
    * increment the "Connection errors" counter
    * stop processing this packet
  - Otherwise:
    * change the state to ESTABLISHED
- ESTABLISHED and FIN-WAIT-2 states:
  - If $SEG.ACK < RCV.UNA$:
    * increment the "Duplicate ACK" counter
    * stop processing this packet

- If $RCV.NXT < SEG.ACK$:
  * check the out-of-sequence queue - if there are any packets from the receiver of this packet to the sender of this packet, see if any of the sequences are $< SEG.ACK$; if so, for each matching packet on the queue, in order, based on their sequence numbers:
    · recursively process that packet and remove it from the queue; if while processing a packet, it finds a $RCV.NXT < SEG.ACK$, then just increment the "Hole" counter, set both RCV.NXT and RCV.UNA to SEG.ACK, and continue (in other words, we will recurse from $SND \rightarrow RCV$ to $RCV \rightarrow SND$, from the perspective of the current segment, but no further).

    Once complete, continue to process this packet.
  * If after processing any out-of-sequence packets, recheck $SEG.ACK < RCV.UNA$; if true:
    · increment the "Wrong ACK" counter
    · stop processing this packet
  * If after processing any out-of-sequence packets, recheck $RCV.NXT < SEG.ACK$; if true:
    · increment the "Hole" counter
    · set RCV.NXT to SEG.ACK
    · set RCV.UNA to SEG.ACK
  * set RCV.UNA to SEG.ACK
  * If $SND.WL1 < SEG.SEQ$ or ($SND.WL1 = SEG.SEQ$ and $SND.WL2 < SEG.ACK$), then set SND.WND to SEG.WND, set SND.WL1 to SEG.SEQ, and set SND.WL2 to SEG.ACK

- FIN-WAIT-1 state:
  - Follow the steps for the ESTABLISHED state
  - If $SEG.ACK = RCV.FIN + 1$, then set the state to FIN-WAIT-2 and set RCV.FIN to zero
- CLOSING state:
  - Follow the steps for the ESTABLISHED state
  - If $SEG.ACK = RCV.FIN + 1$, then set RCV.FIN to zero (this, plus the state, indicates that it has been ACKed) and set the state to TIME-WAIT, otherwise increment the "Disconnection errors" counter
- TIME-WAIT state:
  - Follow the steps for the ESTABLISHED state
  - If $SEG.ACK = RCV.FIN + 1$, then close the record

Note from this point forward that we don't need to handle the SYN-ACK state as it will have either transitioned to the ESTABLISHED state or have stopped processing with an error.

Fifth, check the URG flag:

- ESTABLISHED, FIN-WAIT-1, and FIN-WAIT-2 states:
  - If the URG flag is set, set RCV.UP to the max of RCV.UP and SEG.UP and set RCV.UF to true
  - If RCV.UF is true and $SEG.SEQ <= RCV.UP$, increment the "Urgent" counter, otherwise, set RCV.UF to false
- CLOSING, and TIME-WAIT states:
  - If the URG flag is set, increment the "Disconnection errors" counter

Sixth, update the source's sequence information:

- ESTABLISHED, FIN-WAIT-1, and FIN-WAIT-2 states:
  - If $SEG.LEN > 0$, set SND.NXT to $SEG.SEQ + SEG.LEN$
- CLOSING, and TIME-WAIT states:
  - If $SEG.LEN > 0$, increment the "Wrong data packet size" counter

Finally, check the FIN flag. Since we only process Segment Arrived events, there are only two ways that a connection can be closed:

1. Host A sends a FIN to Host B, Host B ACKs this FIN. Eventually, Host B sends its own FIN, which Host A will ACK, and the connection is CLOSED.
2. Host A sends a FIN to Host B, Host B is thinking the same thing and sends a FIN to Host A. One of the hosts (it does not matter which) ACKs the other's FIN, then the other host ACKs the other FIN, and the connection is CLOSED.

We tweak the event model in RFC 793 to fit these two choices: When a FIN is received, we enter the FIN-WAIT-1 state. If that FIN is ACKed, we enter the FIN-WAIT-2 state. If, instead, the other ACK is received, we enter the CLOSING state. From the FIN-WAIT-2 state, when we receive the other FIN, we enter the TIME-WAIT state. From the CLOSING state, when one of the FINs is ACKed, we enter the TIME-WAIT state. From the TIME-WAIT state, when the other FIN is ACKed, we enter the CLOSED state. This is illustrated in figure 1.

So we perform the following steps, if the FIN bit is set:

- ESTABLISHED state:
  - set SND.FIN to SEG.SEQ
  - set SND.NXT to $SEG.SEQ + SEG.LEN + 1$
  - change the state to FIN-WAIT-1
- FIN-WAIT-1 state:
  - if SND.FIN is set (not zero), increment the "Disconnection errors" counter and stop processing this packet
  - set SND.FIN to SEG.SEQ
  - set SND.NXT to $SEG.SEQ + SEG.LEN + 1$
  - change the state to CLOSING

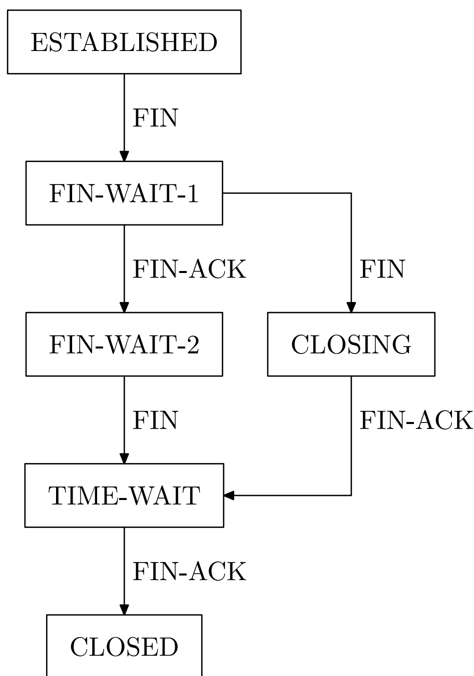

Figure 1: State diagram of how a connection can move from ESTABLISHED to CLOSED.

- FIN-WAIT-2 state:
  - if SND.FIN is set (not zero), increment the "Disconnection errors" counter and stop processing this packet
  - set SND.FIN to SEG.SEQ
  - set SND.NXT to $SEG.SEQ + SEG.LEN + 1$
  - change the state to TIME-WAIT
- CLOSING and TIME-WAIT states:
  - increment the "Disconnection errors" counter
  - stop processing this packet

3.4 Cleaning up

When we are done observing the traffic over this connection, we will still have a number of open connection records. We must decide if these connections are still active, in which case we do not have all the information on the connection to report its metrics, or if the connection has timed out. If a connection has timed out, at least one end of the connection has closed its TCB (or it never created one), and any further attempts to use that connection will result in a

RST (or will just be ignored). As explained in section 2, if the last packet was more than TIMEOUT seconds before we stopped observing traffic on this link, then we will assume the connection is idle and close it.

- If the state of the connection is SYN-RECEIVED, then report the "SYN-ONLY" counter as one, otherwise report it as zero.
- If the state of the connection is SYN-ACK, then report the "SYN-ACK" counter as one, otherwise report it as zero.
- If the state of the connection is ESTABLISHED, then report the "Idle connection" counter as one, otherwise report it as zero.
- If the state of the connection is FIN-WAIT-1, FIN-WAIT-2, CLOSING, or TIME-WAIT, then report the "Half-open connection" counter as one, otherwise report it as zero.

References

Miller, T. (2000, 30 August). Intrusion detection level analysis of Nmap and Queso. Technical Report 1225, SecurityFocus.

Postal, J. (1981, September). RFC 793: Transmission control protocol.