Fault model for HTTP/3 and related network protocols

15-300, Fall 2021

Shannon Ding

December 10, 2021

1 Specific Items

I will be working with Professor Eunsuk Kang from the Institute for Software Research. The project website is located at shannoding.github.io/07400

2 Progress Report

I've completed half of the first technical milestone, which was to read background literature on TCP and mathematically write out an abstract state machine for the protocol.

I started with a general background on TCP. TCP is a protocol primarily used to send packets over the Internet. It ensures integrity of the data being sent by using a handshake with the client to open a connection and sending acknowledgements once data is received. In addition, as it is an Internet protocol, it breaks up data into small packets that each have identifying routing information.

Then, I searched up papers that defined TCP interactions with an abstract state machine and found several papers describing the TCP handshake processes as a transition between several states.

A set of slides from a Computer Networks course taught by Gary Harkin [2] describes the high level states and state transitions available. The transitions are caused by the type of packet sent, either SYN, ACK, or FIN.

1. The server starts in a passively open state, where it will listen for any connections from clients. This is the server's LISTEN state.

2. A client starts a connection by sending a SYN segment. A segment is a piece of data with meta information that is one level above packets. The client transitions to a SYN-SENT state.

3. The server receives the SYN and sends back a SYN and ACK. It then moves into the SYN-RECEIVED state.

4. The client receives the SYN and ACK and sends back an ACK. The client moves into the ESTABLISHED state.

5. The server receives the ACK and moves into the ESTABLISHED state.

6. The client and server can begin exchanging data. When they are finished, one side sends a FIN packet and enters the FIN-WAIT state.

7. The other side frees its resources and replies with an ACK and FIN entering the FIN-WAIT state.

8. Once the first side receives the FIN, it can free its resources and send an ACK, returning to it's initial state.

9. The second side receives an ACK and closes the connection, returning to its initial state.

10. At all stages, there are additional transitions due to timing out before receiving the next segment. The connection can be interrupted at any time with a RST (reset).

The slides also include pseudocode (which is how to define an abstract state machine) for the client and server exchanging data after the handshake. In general, these slides serve as an overview of the protocol, but don't discuss the several different substates a SYN, ACK, or FIN can be in. Furthermore, they don't mention the possible error states and responses.

Treurniet and Lefebvre [3] created a finite state machine (FSM), which is a type of abstract state machine, for TCP's Transport Layer. This paper looked more closely into the contents of packet headers. They identified the type of packet and also the allowed set of flags within each type. Then, they created a FSM with MATLAB to trace the stages of packets and their flags. They collected traffic from 3 Class B networks, which constituted billions of packets. Then, they ran their finite state machine (FSM) on the packets. They found 37% of traffic led to some error state on their FSM. Once malicious activity was removed, the error rate was 4%. Error resulted from receiving or not receiving packets when expected, or receiving packets with bad flags. The authors examined the possible causes of these errors and concluded the main cause was unresponsive hosts either because the packet was too delayed while traveling through the network, the connection was closed early, or an application didn't implement TCP correctly.

Part of the difficulty of creating an ASM for TCP is large possible number of inputs to the machine. Each packet not only has a type, but a set of flags and more header information.

Fortunately, techniques have been developed to create ASMs from complex systems like TCP. Janssen [4] utilized a mapper developed by F. Aarts et al. [5] to reduce the inputs to a smaller set of abstract values. Then, they applied the L* algorithm developed by D. Angluin [6] which can observe external behavior of an entity and develop an ASM, to the reduced TCP. They began with defining a simpler ASM for the L* algorithm to use as a starting point. They then set up an actual testing machine on the network that the learner L* algorithm could interact with. In between the testing machine and the learner, they set up a mapper to turn the highly variable set of inputs into a smaller set of abstract values. Lastly, they ran this setup on several testing machines running different operating systems: Ubuntu 12.10, Ubuntu 10.4, Windows 7, and Windows Vista. Since their mapper quite drastically reduced the set of inputs, the ASMs found for each OS were quite simple and understandable. The author suggested future work to be expanding the set of inputs.

3 Reflection on Initial Plan

There haven't been major changes. Fortunately, since TCP is a well studied and commonly used protocol, there is an abundance of research papers on the subject. In addition, TCP is generally understood as transitioning the server and client through states, so there are papers particularly looking at TCP as an ASM. I didn't reach my milestone goal because I didn't write out the ASM yet. Although there are numerous existing ASM examples, I want to pick a specific aspect of the protocol, not just the packet types, to define transitions for. In addition, I want to include transitions to error states. As a result, I'm going to continue with literature search while focusing on how TCP handles errors. The main suprise is how large an ASM can get just by the addition of one new input value. I will stick with the milestone goals, because after reading Janssen's paper in particular, state diagrams seem more approachable. I have the resources needed for this project.

4 References

[1] Kenneth L. McMillan and Lenore D. Zuck. 2019. Formal specification and testing of QUIC. In Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19). Association for Computing Machinery, New York, NY, USA, 227–240. DOI:https://doi.org/10.114

[2] Gary Harkin. 2004. Computer Networks Spring 2004 Lecture 15. University of Montana, Missoula, MT, USA. https://www.cs.montana.edu/courses/spring2004/440/topics/15-transport/lectures/slideset2.pdf

[3] J. Treurniet and J. H. Lefebvre; 2003; A Finite State Machine Model of TCP Connections in the Transport Layer; DRDC Ottawa TM 2003-139; Defence RD Canada – Ottawa

[4] R. Janssen. Learning a State Diagram of TCP Using Abstraction. Radboud University, 2013.

[5] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, 22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings, volume 6435 of Lecture Notes in Computer Science, pages 188–204. Springer, 2010.

[6] D. Angluin. Learning regular sets from queries and counterexamples. Information and Computation, 75(2):87–106, 1987.