SEE TEXT ONLY

TCP

STACK VALIDATION AT NETFLIX

By Jonathan **Looney**

From the time I started writing TCP code, there has always been a great deal of fear around changes to the TCP stack. This comes from multiple sources, but is rooted in the difficulty of thoroughly testing TCP changes prior to deploying them. This is a challenge Netflix faced as it began investing significant time and energy in optimizing the FreeBSD TCP stack used on its Open Connect Appliances (OCAs), which are the servers that deliver video traffic to Netflix streaming clients. In this article, we will describe the way Netflix removed the fear from TCP development and actually gained the ability to confidently test its changes.

The Challenge

There are many reasons why there has traditionally been fear around making changes to the TCP stack.

One challenge is the code complexity. The TCP stack is quite mature. That maturity brings value, because the TCP stack has shown itself to be reliable over a long period of time (and includes myriad bug fixes to enhance that reliability). But that maturity also brings with it complexity: there is legacy code that may not have been cleaned up, special cases that have been added on over the years, and various branches that can be somewhat obscure. That complexity means it is quite easy to break something unintentionally.

To make the task even more challenging, once you have written your code, it is hard to validate whether things are truly "better" or "worse." (And, in fact, something might be "better" in some way while being "worse" in another way.) For example, let's assume you are trying to fix a problem where the TCP stack will retransmit TCP segments unnecessarily. You may fix that bug (lowering the number of unnecessary retransmissions) but accidentally reduce goodput (the effective data throughput rate to the client application) because you are no longer retransmitting as aggressively. However, given the very wide variation in client behavior and network conditions that exist on the Internet, it is hard to test even a reasonable fraction of the possible cases in a lab.

It is also easy to introduce truly catastrophic bugs without noticing. For example, it is not unheard of for a TCP developer to make a seemingly-innocent change and to find out (usually, some time later) that a timer is no longer arming in a corner-case situation and some TCP connections are becoming stuck. Likewise, we have seen cases where a TCP stack fails to send data when it should.

And, to make this all much more complicated, mistakes are hard to fix. After all, if you break TCP connectivity, you may not even be able to copy a fixed kernel to the device.

For all of these reasons, it is both important to thoroughly test TCP changes and also very hard to do.

The Tools

At Netflix, we use various tools to help us validate TCP changes. During and immediately after development, we may use various unit tests or Packet Drill tests to validate that we are seeing the correct behavior. However, once we move on

to lab testing or field testing, we fundamentally use three tools to help us validate TCP changes, client-reported and server-reported metrics, modular TCP stacks, and the Blackbox Recorder.

Metrics

Once a new TCP stack has reached the point of being a candidate for deployment, we begin to have clients use it. Both our clients and our servers gather detailed statistics about their TCP sessions. This includes information such as goodput, RTT, and the number of retransmissions. The clients also tell us how quickly they got the first chunk of data, how steady the data stream was, and whether they encountered any disruptions to the data stream during playback. (And, of course, if the performance is too far below what the clients are expecting, they will attempt to switch their traffic to an OCA outside of the test. This keeps us from causing too much of a negative impact to client traffic.)

Being able to see metrics from both sides of the connection is a great help in understanding the way the code is functioning. Further, Netflix has a robust data analysis pipeline that allows us to retrieve detailed reports comparing TCP connections that used different TCP stacks and to determine whether the differences are statistically significant. By looking at this data, we are able to detect fairly small differences between two TCP stacks.

However, this still leaves an obvious need: you must be able to compare two TCP stacks in a reliable way. For that, we use modular TCP stacks.

Modular TCP Stacks

Netflix uses the modular TCP stack functionality to run multiple TCP stacks on a single server. This has a number of benefits. First, we can choose to use different TCP stacks for different applications. Second, we can thoroughly test new TCP stacks side-by-side with older TCP stacks. Finally, we can (at least, in theory) deploy fixed versions of TCP stacks without requiring an upgrade to the underlying operating system. (We don't currently regularly make use of on-the-fly TCP stack changes, but we are currently testing the infrastructure to enable that.)

To understand how modular TCP stacks benefit Netflix, it may help to explain the way we now do TCP development at Netflix.

We do our development on a development copy of the TCP code. When we ship a new candidate TCP stack, we copy that code to its own directory and give it a version that is incorporated into its name. Once copied, we try to only update that code to deal with necessary API changes. The modular stack infrastructure in FreeBSD allows us to compile the same code, but use different stack names. It conducts symbol mangling to prevent symbol conflicts between different copies of the same code and lets us install the different versions with different stack names. This lets us literally copy our development code into a new TCP module directory, change one or two Makefile variables, and instantly have a new version of the TCP stack.

(By analogy, you can think of these as being similar to FreeBSD releases. We develop on stable/11 and then copy to the releng/11.0 branch. But we will keep developing on stable/11 and eventually copy it to the releng/11.1 branch. At Netflix, we treat our TCP stacks similarly to this paradigm.)

This ability to compile the code unmodified but with a new TCP stack name is a subtle, but important, feature. This lets us directly track code changes between versions without getting distracted by extraneous, non-functional changes that are there solely to support the module renaming.

When Netflix has a candidate TCP stack, we will deploy the old and new TCP stacks side-by-side on the same OCA. We will then have some number of clients use one or the other of these stacks on that OCA and report their metrics. We can then gather the statistical data and compare the performance of the two stacks.

For the test, it is important that we run the old and new stacks side-by-side on the same OCA. This eliminates many variables that could influence the results. The underlying hardware, operating system, and operating environment should impact the two TCP stacks in exactly the same way. By running the two stacks side-by-side on the same OCA, we are able to focus on just the differences caused by the way the TCP stacks—themselves—perform.

The modular TCP stack functionality also provides a smooth transition to the deployment of the new TCP stack. Once we have validated the new TCP stack on a small-scale test, we move to validate it with gradually larger tests. In our final test, we might have the new TCP stack handle 50% of all Netflix client traffic globally and validate that the stack still performs as expected. If it does, we can switch all clients to use the new stack by default. (And, because the stack can be selected at runtime, it doesn't even require a reboot of the OCAs.)

Using the modular TCP stack functionality, we can also choose the best TCP stack for each application and client. For example, we might find that the new TCP stack performs noticeably worse for a subset of clients. We can set those clients to use the old TCP stack while we investigate that further while we have the rest of the clients benefit from the new TCP stack's improvements. Likewise, we can conduct this testing separately for each application running on our OCAs and change those applications to use the new TCP stack on their own schedule. This lets

us ensure we are using the TCP stack that we've validated best serves the needs of our clients when using each application on our OCAs.

The modular TCP stack functionality also provides us with protection against TCP stack bugs. We have occasionally found very serious bugs in the new TCP stacks. We are able to dynamically unload those modules without interrupting our services. The OCA (and our clients) continue to function using the old TCP stacks installed on the OCA.

Once we fix the bug, we can deploy a new TCP module, load it, and begin testing it. Again, this doesn't require a reboot and is fairly seamless. At the moment, we only regularly use this in development, but we are now testing the tooling we've written to enable us to automate dynamic TCP module deployment across our network of OCAs.

Through the combination of these capabilities, we can have much more confidence deploying our new TCP stacks. We are able to easily create named "release candidate" versions of our TCP stacks, deploy those on OCAs, test them side-by-side with the existing stacks, and—if necessary—recover from a serious TCP stack bug and iterate with new versions that fix bugs in the TCP stack. This is a drastically different picture than the culture of fear described earlier. This is a healthy environment in which to do confident TCP stack development and it is enabled by the ability to easily build, deploy, and use modular TCP stacks.

But, the testing up to this point has focused heavily on examining metrics maintained by the client and server. There is one more aspect to our testing which is worth noting, and that is provided by the TCP Blackbox Recorder.

TCP Blackbox Recorder

The TCP Blackbox Recorder provides a data stream of events from TCP connections. It is named the "Blackbox Recorder" after the flight data recorders (colloquially known as "blackbox recorders") carried on commercial airliners. As initially conceived, the Blackbox Recorder would log a stream of events that occurred on a TCP connection to a ring buffer associated with that connection. If either the user-space application or the kernel notices that something has gone "wrong" with the connection, it can dump out the contents of the ring buffer for later analysis. (And, in the case of kernel panics, a developer can pull data from the ring buffer to see the sequence of events leading up to the crash.)

The events each contain a record of the internal state of the TCP connection so a developer can track how the internal state changed between events. By tracking both internal events and the internal state of the TCP connection, a developer can get a good record of why a connection behaved in the way it did. In fact, this internal information can produce valuable insights that are hidden from analysis tools that only track what was sent and received.

That functionality is very useful and we've used it to debug problems at Netflix. However, the Blackbox Recorder also has another mode of operation that can be useful in finding problems. The Blackbox Recorder allows us to select a percentage of TCP connections for "continuous" logging, where it tries to export all the events from the TCP connection to user space for later analysis. This data can be useful in understanding exactly what has occurred on TCP connections—even connections that we think look "normal."

During development, we may manually scan a sampling of these records to ensure the behavior matches our expectations. We will pay particular attention to both areas we think should have changed due to our code enhancements and also areas we fear might have been accidentally impacted by our code changes. However, there is a limit to how much we can detect through manual review.

We also have an automated tool that scans the Blackbox records to validate that each session operates in keeping with some basic assumptions. For example, we check that we are sending data at least once every second or two when we have no reason to pause (there is window space available and data to send). This lets us validate that timers are working as expected. We also check that we are not exceeding the congestion window and peer's receive window for the connection. This lets us validate that we are not sending data when we should not be doing so. These types of very basic sanity checks are valuable in finding corner-case bugs that escape our lab testing, but are revealed once we start wider testing with real clients.

When this system finds an error, it posts an alert for the TCP development team. They can retrieve the trace and try to determine why the error occurred. And, thanks to the state information logged with the events, it is usually fairly easy to isolate the problem (or, at least, the problematic area of the code).

An Example

It might help to give an example of how this works using a contrived example that is an amalgamation of experiences we have had using this infrastructure and methodology.

For this example, let's assume the current version of the TCP stack we are using is rack_11. The development version is rack_12 but has not changed since rack_11 was created. (In other words, rack_11 and rack_12 are using the same code.) We want to fix a bug with unnecessarily

high retransmissions in rack_11.

We make our code changes to the development version and test it. We think it fixes the problem. We now deploy both rack_11 and rack_12 to an OCA and run a test with a small number of clients. The metrics look good, so we expand the test to cover a few more OCAs and additional clients. At this point, we notice that retransmissions have dropped (which was expected and is "good"). But, we also notice that goodput has dropped (which was not expected and is "bad"). We gather Blackbox traces and find that we accidentally used a <= comparison where we should have used a < comparison. This caused us to not retransmit in some cases where we should have retransmitted.

We fix this bug in the development version and test it again. We think it fixes the new problem (while also still fixing the original problem). We again deploy both rack_11 and rack_12 to an OCA and run a test with a small number of clients. The metrics look good, so we expand the test to cover a few more OCAs and additional clients. The metrics still look good, so we commit the code.

Eventually, rack_12 becomes the release candidate and rack_13 becomes the new development version. At this point, we deploy rack_12 to a larger set of OCAs. The metrics still look good. However, at this point, the Blackbox analysis program alerts that we are failing to send data in a small number of cases. We realize that we have accidentally failed to reset the retransmit timer in a particular corner case.

We fix this new bug in rack_13 (the new development version) and test it again. We deploy both rack_11 and rack_13 to an OCA and run a test with a small number of clients. We then request that our release engineer merge our commit to rack_12. For the sake of our example, we'll say he agrees, so rack_12 now has the new bug fix. We deploy this to the larger set of OCAs and continue our release testing. The metrics still look good, and there are no more alerts from the Blackbox analysis program.

Now, we deploy rack_12 to all OCAs in the Netflix network. We test with a small percentage of clients. The metrics look good, so we proceed with testing against 50% of all Netflix clients. At this point, we notice that the metrics for one client operating system are poor, while the metrics for all others are the same or better. We decide to direct all Netflix clients to use the new TCP stack, but we make an exception for the one operating system with worse metrics and tell the clients using that operating system to use the old TCP stack.

We then begin to gather focused testing and debugging data for that one client type in an effort to understand why it performs more poorly with the new TCP stack code than with the old code. Hopefully, we are able to fix the bug in rack_13. At that point, we start the testing process again.

In reality, this process sometimes takes longer than we would like. For example, we have deployed release candidates across all OCAs in the Netflix network, only to abandon the release candidate in a late stage of testing due to a subtle metric change that escaped detection during earlier, more narrow testing phases. But, even if it sometimes takes longer than we would like, we are happy with the result this thorough testing process produces: the ability to confidently upgrade the code in our TCP stacks.

TCP Development with Confidence

Given all these tools, we are able to conduct TCP development with a greater degree of confidence that we are making things better. We know that we will not write bug-free code. We know that we will not catch all the problems in a lab. But we can slowly and carefully deploy TCP stack changes and test them in a way that ensures we can test TCP stack changes without having our results influenced by differences in the underlying hardware or operating system. And we can validate that the new TCP stack behaves correctly both by looking at server-side and client-side metrics and also by examining traces of the internal operation of the TCP stacks.

The combination of these things has changed the TCP development paradigm from one constrained by fear to one where we have great freedom to innovate, confident that we have the right tools to innovate responsibly.

Note: Much of the code (including the RACK TCP stack) described in this article is already available in FreeBSD 12. A few things (such as enhanced server-side stats and the user-space Blackbox analysis tools) are still in the process of being upstreamed, but should land "soon." ●

Jonathan Looney manages a development team at Netflix responsible for maintaining the operating system that runs on the OCAs. He is a FreeBSD committer active in the transport protocols area.