This article helps you understand ways to improve connection latency between processes within Google Cloud. The article outlines methods to compute correct settings for decreasing the latency of TCP connection.

Modern microservices architecture advocates that developers should build small services with single responsibility. The services should communicate using TCP or UDP, based on the reliability expectations of the system. It's therefore critical for microservices-based systems to communicate with reliability and low latency.

Google Cloud provides both reliability and low latency by providing a global network (/about/locations/#meet-our-network), which means that our users can also go global. Having a global network means that a user can simply create a VPC (/vpc/) that can span regions and zones (/compute/docs/regions-zones/). Applications can connect to each other across regions and zones without ever leaving the Google Cloud network.

Applications that have been written for a traditional data center environment can exhibit slow performance when they're moved to a hybrid cloud environment—that is, when some of the application components run in a corporate data center and others run in the cloud. Slow performance can be the result of a number of factors. This article focuses on round-trip latencies and how latency affects TCP performance in applications that move a considerable amount of data over any part of the network.

TCP uses a windowing mechanism to prevent a fast sender from overrunning a slow receiver. The receiver advertises how much data the sender should send before the sender must wait for a window update from the receiver. As a result, if a receiving application can't receive data on the connection, there's a limit to how much data can be queued waiting for the application.

The TCP window allows efficient use of memory on the sending and receiving systems. As the receiving application consumes data, window updates are sent to the sender. The fastest that the window update can happen is in one round trip, which leads to the following formula for one of the limits to the bulk transfer performance of a TCP connection:

Throughput <= window size / round-trip time (RTT) latency

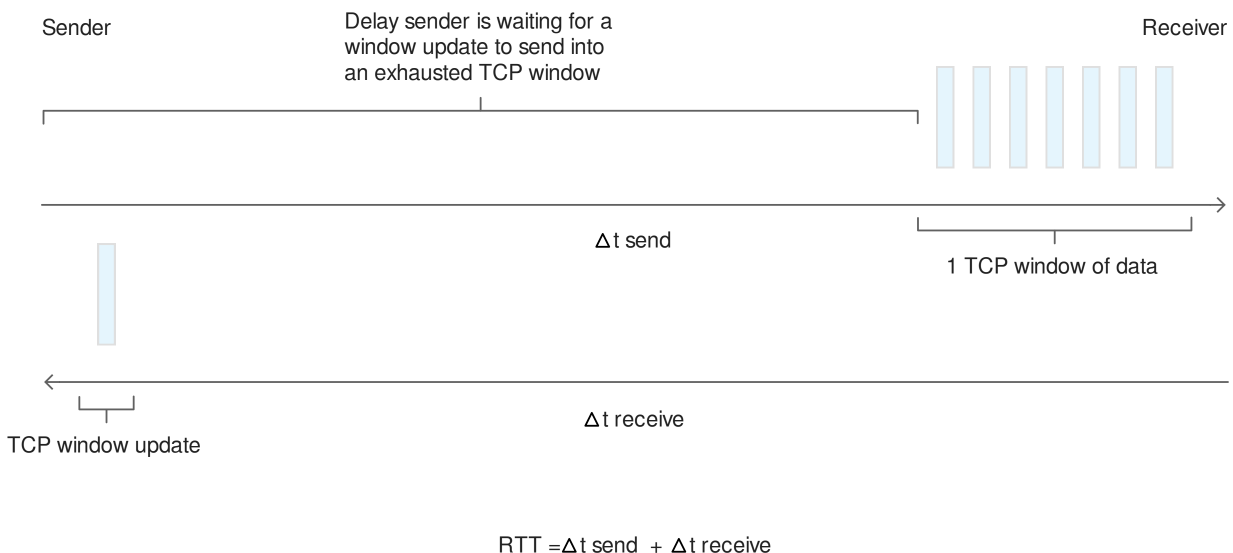
In the original design for TCP, this window has a maximum size of 65535 bytes (64 KiB - 1). This was the maximum amount of data that the sender could send before the sender received a window update in order to allow more data to be sent.

Since TCP was introduced, some key features have changed:

- Typical network speeds have increased by four orders of magnitude.
- Typical memory in a system has increased by four orders of magnitude.

The result of the first change is that the original TCP window sizes led to an inefficient use of network resources. A sender would send a window's worth of data at the best speed possible under network conditions, and then sit idle for a considerable length of time while waiting for the TCP window update. The result of the second change is that senders and receivers can use more memory for networking to address the limitation exposed by first change.

The following diagram illustrates this interchange.




The sender can't fully utilize the network, because it's waiting for the TCP window update before sending additional data.

The solution is to send more data at a time. As the bandwidth of the network increases, more data can fit into the pipe (network), and as the pipe gets longer, it takes longer to acknowledge the receipt of the data. This relationship is known as the bandwidth-delay product (BDP). This is calculated as the bandwidth multiplied by the round-trip time (RTT), resulting in a value that specifies the optimal number of bits to send in order to fill the pipe. The formula is this:

```
BDP (bits) = bandwidth (bits/second) * RTT (seconds)
```

Computed BDP is used as TCP window size for optimization.

For example, imagine that you have a 10 Gbps network with an RTT of 30 milliseconds. For the window size, use the value of the original TCP window size (65535 bytes). This value doesn't come close to taking advantage of the bandwidth capability. The maximum TCP performance possible on this link is as follows:

```
(65535 bytes * 8 bits/byte) = bandwidth * 0.030 second
bandwidth = (65535 bytes * 8 bits/byte) / 0.030 second
bandwidth = 524280 bits / 0.030 second
bandwidth = 17476000 bits / second
```

To state it another way, these values result in throughput that's a bit more than 17 Mbits per second, which is a small fraction of network's 10 Gbps capability.

To resolve the performance limitations imposed by the original design of TCP window size, extensions to the TCP protocol were introduced that allow the window size to be scaled to much larger values. Window scaling supports windows up to 1,073,725,440 bytes, or almost 1 GiB. This feature is outlined in RFC 1323 (https://tools.ietf.org/html/rfc1323) as TCP window scale option (https://tools.ietf.org/html/rfc1323#page-8).

The window scale extensions expand the definition of the TCP window to use 32 bits, and then use a scale factor to carry this 32-bit value in the 16-bit window field of the TCP header. To see if the feature is enabled on Linux-based systems, use the following command:

(All Google Cloud Linux virtual machines have this feature enabled by default.) A return value of `1` indicates that the option is enabled. If the feature is disabled, you can enable it by using the following command:

You can use the previous example to show the benefit of having window scaling. As before, assume a 10 Gbps network with 30-millisecond latency, and then compute a new window size using this formula:

```
(Link speed * latency) / 8 bits = window size
```

If you plug in the example numbers, you get this:

```
(10 Gbps * 30ms/1000sec) / 8bits/byte = window size
(10000 Mbps * 0.030 second) / 8 bits/byte = 37.5 MB
```

Increasing the TCP window size to 37 MB can increase the theoretical limit of TCP bulk transfer performance to a value approaching the network capability. Of course, many other factors can limit performance, including system overhead, average packet size, and number of other flows sharing the link, but as you can see, the window size substantially mitigates the limits imposed by the previous limited window size.

In Linux, the TCP window size is affected by the following `sysctl(8)` tunables:

The first two tunables affect the maximum TCP window size for applications that attempt to control the TCP window size directly, by limiting the applications' request to no more than those values. The second two tunables affect the TCP window size for applications that let Linux auto-tuning do the work.

The optimal window-size value depends on your specific circumstances, but one starting point is the largest BDP (bandwidth-delay product) for the path or paths over which you expect the system to send data. In that case, you want to set the tunables by using following steps:

1. Make sure that you have root privileges.
2. Get the current buffer settings. Save these settings in case you want to roll back these changes.
3. Set an environment variable to the new TCP window size that you want to use:
4. Set the maximum OS receive buffer size for all types of connections:

5. Set the maximum OS send buffer size for all types of connections:

6. Set the TCP receive memory buffer (`tcp_rmem`) settings:

   The `tcp_rmem` setting takes three values:

   - The minimum receive buffer size that can be allocated for a TCP socket. In this example, the value is `4096` bytes.
   - The default receive buffer size, which also overrides the `/proc/sys/net/core/rmem_default` value used by other protocols. In the example, the value is `87380` bytes.
   - The maximum receive buffer size that can be allocated for a TCP socket. In the example, this is set to the value that you set earlier (`8388608` bytes).

7. Set the TCP send memory buffer (`tcp_wmem`) settings:

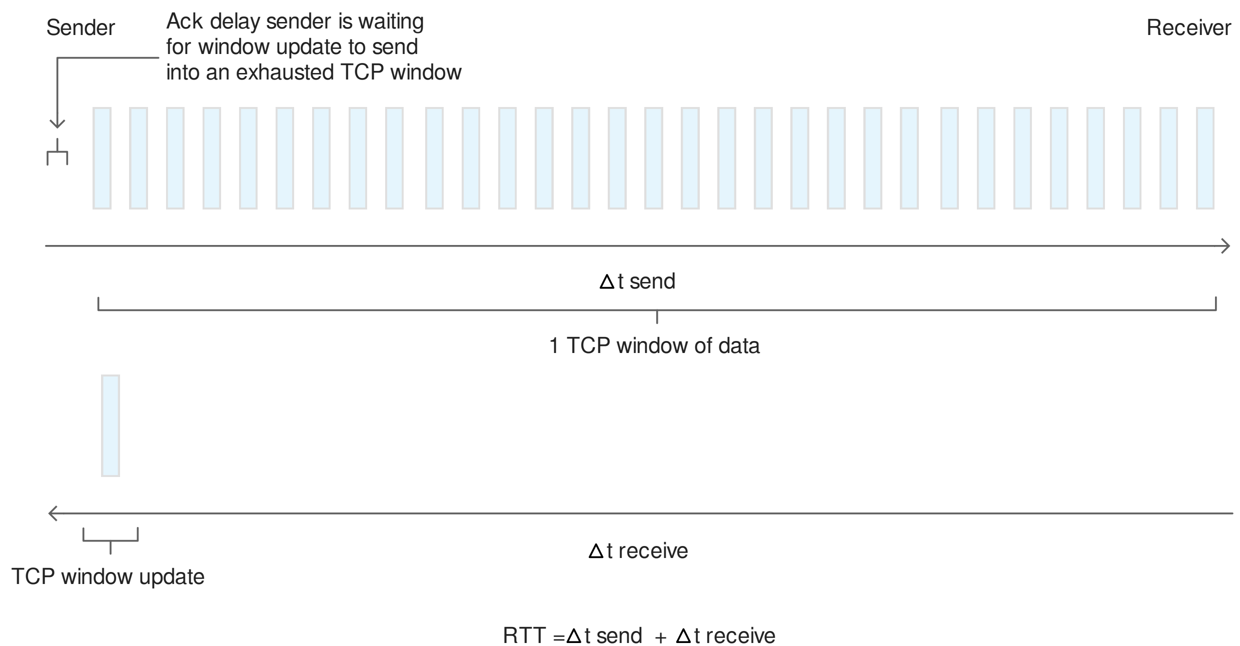
   The `tcp_wmem` setting takes three values:

   - The minimum TCP send buffer space available for a single TCP socket.
   - The default buffer space allowed for a single TCP socket.
   - The maximum TCP send buffer space.

8. Set the tunables so that subsequent connections use the values you specified:

To persist these settings across reboots, append the commands you set previously to the `/etc/sysctl.conf` file:

When TCP has a large enough window size to utilize the BDP, the picture changes, as shown in the following diagram:




The TCP window size can always be adapted based on the resources available to the process involved and the TCP algorithm in use. As the diagram shows, window scaling lets a

connection go well beyond the 65 KiB window size defined in original TCP specification.

You can test this yourself. First, make sure that you've made TCP window size changes to your local computer and to a remote computer by setting the tunables on both machines. Then run the following commands:.

The first command creates a 1 GB `sample.txt` file that has random data. The second command copies that file from your local machine to a remote machine.

Note the `scp` command output on the console, which displays bandwidth in Kbps. You should see sizable difference in the results from before and after the TCP window size changes.

- Read the blog post on 5 steps to better Google Cloud networking performance (/blog/products/gcp/5-steps-to-better-gcp-network-performance?hl=ml).
- Learn about Global Networking Products (/products/networking/).
- Read more about Networking Tiers (/blog/products/gcp/introducing-network-service-tiers-your-cloud-network-your-way) on Google Cloud.
- Learn about Iperf (https://wikipedia.org/wiki/Iperf), a commonly used network testing tool that can create TCP/UDP data streams and measure the throughput of the network that carries them.
- Learn to use Netperf (https://hewlettpackard.github.io/netperf/), another good network testing tool, which is also used by the PerfKitBenchmark (https://github.com/GoogleCloudPlatform/PerfKitBenchmarker) suite to test performance and benchmark the various cloud providers against one another.

- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).