**Performing in the cloud – network latency**

To me, ‘cloud computing’ is renting a compute resource to perform a task. In order to use that compute resource, you need to instruct it to do something, which is typically done via the network. If the task the compute resource needs to fulfil is being an application server or being a client or both in the case of an application server that uses an Oracle database, the network latency between the client of the database and the database server is a critical property.

I think so far everybody is with me. If we zoom in to the network, it becomes more difficult, and \*very\* easy to make wrong assumptions. Let me explain. A network, but really any connection between processing and a resource, has two DIFFERENT properties that I see getting mixed up consistently. These are:  
\* Latency: the time it takes for a signal or (network) packet to travel from the client to the server, or the time it takes to travel from the client to the server and back.  
\* Bandwidth: the amount of data that can be transported from the client to the server in a certain time.

How do you determine the latency of a network? Probably the most people respond with ‘use ping’. This is how that looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [opc@oid1 ~]$ ping -c 3 100.65.22.62  PING 100.65.22.62 (100.65.22.62) 56(84) bytes of data.  64 bytes from 100.65.22.62: icmp\_seq=1 ttl=62 time=680 ms  64 bytes from 100.65.22.62: icmp\_seq=2 ttl=62 time=0.304 ms  64 bytes from 100.65.22.62: icmp\_seq=3 ttl=62 time=0.286 ms |

The question I often ask myself is: what is that we see actually? How does this work?  
In order to answer that question, the tcpdump tool can answer that question. Using tcpdump, you can capture the network packets on which the ping utility based the above outcome. The ‘-ttt’ option calculates the time between each arrived packet:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | [opc@oid1 ~]$ sudo tcpdump -ttt -i any host 100.65.22.62  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on any, link-type LINUX\_SLL (Linux cooked), capture size 65535 bytes  00:00:00.000000 IP oid1.oraclecloud.com > lsh1.oraclecloud.com: ICMP echo request, id 35879, seq 1, length 64  00:00:00.680289 IP lsh1.oraclecloud.com > oid1.oraclecloud.com: ICMP echo reply, id 35879, seq 1, length 64  00:00:00.319614 IP oid1.oraclecloud.com > lsh1.oraclecloud.com: ICMP echo request, id 35879, seq 2, length 64  00:00:00.000287 IP lsh1.oraclecloud.com > oid1.oraclecloud.com: ICMP echo reply, id 35879, seq 2, length 64  00:00:01.000180 IP oid1.oraclecloud.com > lsh1.oraclecloud.com: ICMP echo request, id 35879, seq 3, length 64  00:00:00.000269 IP lsh1.oraclecloud.com > oid1.oraclecloud.com: ICMP echo reply, id 35879, seq 3, length 64 |

So, ping works by sending a packet (ICMP echo request) requesting a reply (ICMP echo reply) from the remote server, and measure the time it takes to get that reply. Great, quite simple, isn’t it? However, the biggest issue I see this is using a protocol that is not used for sending regular data (!). Most application servers I encounter send data using TCP (transmission control protocol), the traffic ping sends are sent using a protocol called ICMP (internet control message protocol). Especially in the cloud, which means (probably) a lot of the infrastructure is shared, ICMP might be given different priority than TCP traffic, which you quite probably are using when the application on your cloud virtual machine is running. For those of you who haven’t looked into the network side of the IT landscape, you can priorise protocols and even specific ports, throttle traffic and you can even terminate it. In fact, a sensible protected (virtual) machine in the cloud will not respond to ICMP echo requests in order to protected it from attacks.

So, what would be a more sensible approach then? A better way would be to use the same protocol and port number that your application is going to use. This can be done using a tool called [hping](https://centos.pkgs.org/6/epel-x86_64/hping3-0.0.20051105-16.el6.x86_64.rpm.html). Using that tool, you can craft your own packet with the protocol and flags you want. In the case of Oracle database traffic that would be the TCP protocol, port 1521 (it can be any port number, 1521 is the default port). This is how you can do that. In order to mimic starting a connection, the S (SYN) flag is set (-S), one packet is send (-c 1) to port 1521 (-p 1521).

|  |  |
| --- | --- |
| 1 | [opc@oid1 ~]$ sudo hping -S -c 1 -p 1521 129.152.192.2 |

What this does is best investigated with tcpdump once again. The server this is executed against can respond in two ways (three actually). When you send this to TCP port 1521 where a listener (or any other daemon that listens on that port) is listening, this is the response:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [opc@oid1 ~]$ sudo tcpdump -ttt -i any host 129.152.192.3  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on any, link-type LINUX\_SLL (Linux cooked), capture size 65535 bytes  00:00:00.000000 IP oid1.oraclecloud.com.kjtsiteserver > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [S], seq 1436552830, win 512, length 0  00:00:00.001229 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.kjtsiteserver: Flags [S.], seq 2397022511, ack 1436552831, win 14600, options [mss 1460], length 0  00:00:00.000023 IP oid1.oraclecloud.com.kjtsiteserver > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [R], seq 1436552831, win 0, length 0 |

This is a variation of the classic TCP three way handshake:  
1. A TCP packet is sent with the SYN flag set to indicate starting a (client to server) connection.  
2. A TCP packet is sent back with SYN flag set to indicate starting a (server to client) connection, and the first packet is acknowledged.  
3. This is where the variation is, normally an acknowledgement would be sent of the second packet to establish a two way connection, but in order to stop the communication a packet is sent with the RST (reset) flag set.

However, this is if a process is listening on the port. This is how that looks like when there is no process listening on port 1521:

|  |  |
| --- | --- |
| 1  2  3  4  5 | [opc@oid1 ~]$ sudo tcpdump -ttt -i any host 129.152.192.2  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on any, link-type LINUX\_SLL (Linux cooked), capture size 65535 bytes  00:00:00.000000 IP oid1.oraclecloud.com.vsamredirector > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [S], seq 1975471906, win 512, length 0  00:00:00.001118 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.vsamredirector: Flags [R.], seq 0, ack 1975471907, win 0, length 0 |

This means that if a connection is initiated to a port on which no process is listening (port status ‘closed’), there is communication between the client and the server. This is why firewalls were invented!  
1. A TCP packet is sent with the SYN flag set to indicate starting a connection.  
2. A TCP packet is sent back to with the RST (reset) flag set to indicate no connection is possible.

The third option, when port 1521 is firewalled on the server, simply means only the first packet (from client to server with the SYN flag set) is sent and no response is coming back.

Okay, let’s pick up the performance aspect again. This hping command:

|  |  |
| --- | --- |
| 1  2  3 | [opc@oid1 ~]$ sudo hping -S -c 1 -p 1521 129.152.192.3  HPING 129.152.192.3 (eth0 129.152.192.3): S set, 40 headers + 0 data bytes  len=44 ip=129.152.192.3 ttl=57 DF id=0 sport=1521 flags=SA seq=0 win=14600 rtt=1.2 ms |

Says the roundtrip time is 1.2ms. If we look at the network packets and timing:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [opc@oid1 ~]$ sudo tcpdump -ttt -i any host 129.152.192.3  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on any, link-type LINUX\_SLL (Linux cooked), capture size 65535 bytes  00:00:00.000000 IP oid1.oraclecloud.com.mmcal > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [S], seq 1289836562, win 512, length 0  00:00:00.001113 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.mmcal: Flags [S.], seq 2504750542, ack 1289836563, win 14600, options [mss 1460], length 0  00:00:00.000016 IP oid1.oraclecloud.com.mmcal > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [R], seq 1289836563, win 0, length 0 |

It becomes apparent that the 1.2ms time hping reports is the time it takes for the remote server to send back the SYN+ACK package in the TCP three way handshake.

So does that mean that if we take a number of measurements (let’s say 100, or 1000) to have a statistically significant number of measurements we can establish my TCP roundtrip time and then know how fast my connection will be (outside of all the other variables inherent to the internet and potential noisy neighbours to name a few)?

Oracle provides a way to generate and measure SQL-Net traffic in My Oracle Support note: Measuring Network Capacity using oratcptest (Doc ID 2064368.1). This note provides a jar file which contains server and client software, and is aimed at dataguard, but is useful to measure SQL-Net network latency. I have looked at the packets oratcptest generates, and they mimic SQL-Net quite well.

Let’s see if we can redo the test above to measure pure network latency. First on the database server side, setup the server:

|  |  |
| --- | --- |
| 1 | [opc@cfcldv0539m ~]$ java -jar oratcptest.jar -server 129.152.192.2 -port=1521 |

And then on the client side run the client using the same oratcptest jar file:

|  |  |
| --- | --- |
| 1 | java -jar oratcptest.jar 129.152.192.2 -mode=sync -length=0 -duration=1s -interval=1s -port=1521 |

The important bits are -mode=sync (client packet must be acknowledged before sending another packet) and -length=0 (network traffic contains no payload). This is the result:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | [Requesting a test]      Message payload        = 0 bytes      Payload content type   = RANDOM      Delay between messages = NO      Number of connections  = 1      Socket send buffer     = (system default)      Transport mode         = SYNC      Disk write             = NO      Statistics interval    = 1 second      Test duration          = 1 second      Test frequency         = NO      Network Timeout        = NO      (1 Mbyte = 1024x1024 bytes)    (07:34:42) The server is ready.                          Throughput                 Latency  (07:34:43)          0.017 Mbytes/s                0.670 ms  (07:34:43) Test finished.             Socket send buffer = 11700 bytes                Avg. throughput = 0.017 Mbytes/s                   Avg. latency = 0.670 ms |

If you look at the hping roundtrip time (1.2ms) and the oratcptest roundtrip time (0.7ms) clearly this is different! If you just look at the numbers (1.2 versus 0.7) it might seem like the oratcptest time is only measuring client to server traffic instead of the whole roundtrip? For this too it’s good to use tcpdump once again and look what oratcptest actually is doing:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | [opc@oid1 ~]$ sudo tcpdump -ttt -i any host 129.152.192.2  tcpdump: verbose output suppressed, use -v or -vv for full protocol decode  listening on any, link-type LINUX\_SLL (Linux cooked), capture size 65535 bytes  00:00:00.000000 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [S], seq 2408800085, win 17920, options [mss 8960,sackOK,TS val 3861246405 ecr 0,nop,wscale 7], length 0  00:00:00.001160 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [S.], seq 2178995555, ack 2408800086, win 14600, options [mss 1460,nop,nop,sackOK,nop,wscale 7], length 0  00:00:00.000015 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [.], ack 1, win 140, length 0  00:00:00.023175 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 1:145, ack 1, win 140, length 144  00:00:00.000520 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [.], ack 145, win 123, length 0  00:00:00.000951 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [P.], seq 1:145, ack 145, win 123, length 144  00:00:00.000008 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [.], ack 145, win 149, length 0  00:00:00.018839 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 145:157, ack 145, win 149, length 12  00:00:00.000563 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [P.], seq 145:149, ack 157, win 123, length 4  00:00:00.000358 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 157:169, ack 149, win 149, length 12  00:00:00.000486 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [P.], seq 149:153, ack 169, win 123, length 4  00:00:00.000100 IP oid1.oraclecloud.com.63602 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 169:181, ack 153, win 149, length 12  00:00:00.000494 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63602: Flags [P.], seq 153:157, ack 181, win 123, length 4  ...  00:00:00.000192 IP oid1.oraclecloud.com.63586 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 18181:18193, ack 6157, win 149, length 12  00:00:00.000447 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63586: Flags [P.], seq 6157:6161, ack 18193, win 123, length 4  00:00:00.006696 IP oid1.oraclecloud.com.63586 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [F.], seq 18193, ack 6161, win 149, length 0  00:00:00.000995 IP cfcldv0539.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.63586: Flags [F.], seq 6161, ack 18194, win 123, length 0  00:00:00.000012 IP oid1.oraclecloud.com.63586 > cfcldv0539.us2.oraclecloud.com.ncube-lm: Flags [.], ack 6162, win 149, length 0 |

If you look at rows 4, 5 and 6 you see the typical TCP three-way handshake. What is nice to see, is that the actual response or roundtrip time for the packet from the server on line 5 actually took 1.1ms, which is what we have measured with hping! At lines 7-10 we see there is a packet send from the client to the server which is ACK’ed and a packet send from the server to the client which is ACK’ed. If you add the ‘-A’ flag to tcpdump you can get the values in the packet printed as characters, which shows the client telling the server how it wants to perform the test and the server responding with the requested settings. This is all a preparation for the test.

Starting from line 11, there is a strict repeating sequence of the client sending a packet of length 12, ACK’ing the previous received packet, and then the server responding with a packet of length 4 ACK’ing its previous received packet. This is the actual performance test! This means that the setting ‘-duration=1s -interval=1s’ does not mean it sends one packet, it actually means it’s continuously sending packets for the duration of 1 second. Also another flag is showing: the P or PSH (push) flag. This flag means the kernel/tcpip-stack understands all data to transmit is provided from ‘userland’, and now must be sent immediately, and instructs the receiving side to process it immediately in order to bring it to the receiving userland application as soon as possible too.

Lines 20-22 show how the connection is closed by sending a packet with a FIN flag, which is done for both the client to the server and the server to the client, and because it’s TCP, these need to be ACK’ed, which is why you see a trailing packet without a flag set, only ACK’ing the FIN packet.

The conclusion so far is that for real usable latency calculations you should not use a different protocol (so whilst ICMP (ping) does give an latency indication it should really only be used as an indicator), and that you should measure doing the actual work, not meta-transactions like the TCP three way handshake. Probably because of the PSH flag, the actual minimal latency for SQL-Net traffic is lower than ping and hping showed.

Wait a minute…did you notice the ‘actual **minimal** latency’? So far we only have been sending empty packets, which means we measured how fast a packet can travel from client to server and back. In reality, you probably want to send actual data back and forth, don’t you? That is something that we actually have not measured yet!

Let’s do actual Oracle transactions. For the sake of testing network latency, we can use [Swingbench](http://dominicgiles.com/swingbench.html)to execute SQL. This is how that is done:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [opc@oid1 bin]$ cd ~/sw/swingbench/bin  [opc@oid1 bin]$ ./charbench -c ../configs/stresstest.xml -u soe -p soe -uc 1 -rt 00:01  Author  :    Dominic Giles  Version :    2.5.0.971    Results will be written to results.xml.  Hit Return to Terminate Run...    Time        Users   TPM TPS    8:22:56 AM      1       14450   775 |

Please mind I am using 1 user (-uc 1) and a testing time of 1 minute (-rt 00:01), which should be longer when you are doing real testing. As a reminder, I am using 1 session because I want to understand the latency, not the bandwidth! In order to understand if the network traffic looks the same as oratcptest.jar, I can use tcpdump once again. Here is a snippet of the traffic:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 00:00:00.000106 IP oid1.oraclecloud.com.50553 > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 5839:5852, ack 5986, win 272, length 13  00:00:00.000491 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.50553: Flags [P.], seq 5986:6001, ack 5852, win 330, length 15  00:00:00.000234 IP oid1.oraclecloud.com.50553 > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 5852:6003, ack 6001, win 272, length 151  00:00:00.000562 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.50553: Flags [P.], seq 6001:6077, ack 6003, win 330, length 76  00:00:00.000098 IP oid1.oraclecloud.com.50553 > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 6003:6016, ack 6077, win 272, length 13  00:00:00.000484 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.50553: Flags [P.], seq 6077:6092, ack 6016, win 330, length 15  00:00:00.000238 IP oid1.oraclecloud.com.50553 > cfcldv0539-vip.us2.oraclecloud.com.ncube-lm: Flags [P.], seq 6016:6159, ack 6092, win 272, length 143  00:00:00.000591 IP cfcldv0539-vip.us2.oraclecloud.com.ncube-lm > oid1.oraclecloud.com.50553: Flags [P.], seq 6092:6425, ack 6159, win 330, length 333 |

The important bit is this shows the same single packet traffic client to server and back as we saw oratcptest generated, however now with varying packet size (which is logical, different SQL statements are sent to the database), the PSH bit is set, which also is the same as oratcptest generated.

Let’s assume this is a real-life workload. In order to measure and calculate differences in performance between different networks, we need the average packet length. This can be done with a tool called [tcpstat (this link provides the EL6 version)](http://www.mirrorservice.org/sites/apt.sw.be/redhat/el6/en/x86_64/rpmforge/RPMS/tcpstat-1.5-2.el6.rf.x86_64.rpm). In my case I have only one application using a database on this server, so I can just filter on port 1521 to measure my SQL-Net traffic:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [opc@oid1 ~]$ sudo tcpstat -i eth0 -o "Packet/s=%p\tmin size: %m\tavg size: %a\tmax size: %M\tstddev: %d\n" -f 'port 1521'  Packet/s=2526.40    min size: 53    avg size: 227.76    max size: 1436  stddev: 289.21  Packet/s=2531.40    min size: 53    avg size: 229.79    max size: 1432  stddev: 291.22  Packet/s=2634.20    min size: 53    avg size: 229.59    max size: 1432  stddev: 293.38  Packet/s=2550.00    min size: 53    avg size: 234.11    max size: 1435  stddev: 296.77  Packet/s=2486.80    min size: 53    avg size: 232.24    max size: 1436  stddev: 293.16 |

In case you wondered why tcpstat reports a minimum length of 53 and tcpdump (a little up in the article) of 13; tcpstat reports full packet length including packet, protocol and frame headers, tcpdump in this case reports the payload length.

Now we can execute oratcptest.jar again, but with a payload size set that matches the average size that we measured, I have taken 250 as payload size:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | [opc@oid1 ~]$ java -jar oratcptest.jar 129.152.192.2 -mode=sync -length=250 -duration=1s -interval=1s -port=1521  [Requesting a test]      Message payload        = 250 bytes      Payload content type   = RANDOM      Delay between messages = NO      Number of connections  = 1      Socket send buffer     = (system default)      Transport mode         = SYNC      Disk write             = NO      Statistics interval    = 1 second      Test duration          = 1 second      Test frequency         = NO      Network Timeout        = NO      (1 Mbyte = 1024x1024 bytes)    (09:39:47) The server is ready.                          Throughput                 Latency  (09:39:48)          0.365 Mbytes/s                0.685 ms  (09:39:48) Test finished.             Socket send buffer = 11700 bytes                Avg. throughput = 0.365 Mbytes/s                   Avg. latency = 0.685 ms |

As you can see, there is a real modest increase in average latency going from 0.670ms to 0.685ms.

In order to test the impact of network latency let’s move the oratcptest client to the server, to get the lowest possible latency. Actually, this is very easy, because the oratcptest.jar file contains both the client and the server, so all I need to do is logon to the server where I started the oratcptest.jar file in server mode, and run it in client mode:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | [opc@cfcldv0539m ~]$ java -jar oratcptest.jar 129.152.192.2 -mode=sync -length=250 -duration=1s -interval=1s -port=1521  [Requesting a test]      Message payload        = 250 bytes      Payload content type   = RANDOM      Delay between messages = NO      Number of connections  = 1      Socket send buffer     = (system default)      Transport mode         = SYNC      Disk write             = NO      Statistics interval    = 1 second      Test duration          = 1 second      Test frequency         = NO      Network Timeout        = NO      (1 Mbyte = 1024x1024 bytes)    (14:49:29) The server is ready.                          Throughput                 Latency  (14:49:30)         12.221 Mbytes/s                0.020 ms  (14:49:30) Test finished.             Socket send buffer = 26010 bytes                Avg. throughput = 11.970 Mbytes/s                   Avg. latency = 0.021 ms |

Wow! The roundtrip latency dropped from 0.685ms to 0.021ms! Another test using oratcptest.jar using a true local network connection (with Linux being virtualised using Xen/OVM) shows a latency of 0.161ms.

These are the different network latency figures measured with oratcptest using a payload size that equals my average network payload size:  
– Local only RTT: 0.021  
– Local network RTT: 0.161  
– Cloud to cloud RTT: 0.685  
If I take swingbench and execute the ‘stresstest’ run local, on a machine directly connected via the network and in another cloud/network, and now measure TPS (transactions per second), I get the following figures:  
– Local only TPS: 2356  
– Local network TPS: 1567  
– Cloud to cloud TPS: 854  
Do these figures make sense?  
– Local only: Time not in network transit per second: 1000-(0.021\*2356)=950.524; approximate average time spend on query: 950.523/2356=0.40ms  
– Local network: 1000-(0.161\*1567)=747.713/1567=0.48ms  
– Cloud to cloud: 1000-(0.685\*854)=415.010/854=0.49ms  
It seems that this swingbench test spends roughly 0.40-0.50ms on processing, the difference in transactions per second seem to be mainly caused by the difference in network latency.