Show Submission Credentials

P3. Profiling a Cloud Service Profile and locate the bottlenecks of a Cloud service

14 days 0 hours left

Introduction and Concepts

Scenario

You are a software engineer at a startup called CCbnb. The company offers web and mobile applications for hotel reservations. You are in charge of developing and maintaining a backend service to query, add, remove or modify reservations. However, as the company is new and the funding is insufficient, your manager has given you a limited budget to build a cloud service that is capable of handling the workload generated by CCbnb users.

In your current web service cluster architecture, there are 3 instances running, but the throughput is relatively low. Even for simple queries like retrieving a record with a key (eg. reservation for a user), it takes 500 ms to respond. Now you want to figure out the *bottleneck* that is slowing down the whole cloud service and whether there is *efficient utilization* of all system resources.

In this primer, we will describe key concepts in profiling to help you with this effort. We will then walk you through ways to profile each component in your system, and eventually describe several methodologies to profile the whole system.

Tips

To locate a bottleneck of a complex system, it’s important to know the capacity of each component and their run-time usages. For example, if you observe that the disk I/O is running at 5% of its capacity with CPU usage at 95%, it’s obvious that the system is slowed down by a task that is computationally bound.

Concepts

> In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. -- Wikipedia (https://en.wikipedia.org/wiki/Profiling_(computer_programming))

We will describe and provide short examples of three profiling techniques below.

Event-based Profiler

Event-based profilers (also called tracing profilers (https://iobservable.net/blog/2013/02/03/profiler-types-and-their-overhead/)) collect data from predefined events including events of entering or leaving a function, an object allocation, or due to exceptions. In Java, the JVMTI (JVM Tools Interface) API provides hooks for trapping events like calls, class load/unload, etc. A simple Python example of set.setprofile can be found below. In this example, we can see that the profiler function is called whenever there’s a function call, c_call, return, or c_return event.

```
#!/usr/bin/env python3
import sys


def profiler(frame, event, arg):
    print(frame, event, arg)


def f():
    print("Welcome to CCbnb!")

sys.setprofile(profiler)
f()
```

The output is:

```
<frame object at 0x7ff1cd702668> call None
<frame object at 0x7ff1cd702668> c_call <built-in function print>
Welcome to CCbnb!
<frame object at 0x7ff1cd702668> c_return <built-in function print>
<frame object at 0x7ff1cd702668> return None
<frame object at 0x7ff1cd702168> return None
<frame object at 0x10bbc6ba8> call None
<frame object at 0x10bbc6ba8> return None
```

Another handy Python profiler based on events is cProfile and its tutorial can be found here (https://docs.python.org/3/library/profile.html#module-cProfile).

Statistical Profiler

Statistical profilers take samples of the target's address space at regular intervals during execution. This technique is a quick way for us to learn which functions are consuming the largest proportions of cycles (https://web.archive.org/web/20210114141352/http://processors.wiki.ti.com/index.php/Statistical_Profiling). However, it is less accurate than the numerical method. As a first step, we can find out which functions consume the largest proportions of cycles. Then, we can use a different profiling technique on these specific functions in order to learn more about the root cause(s).

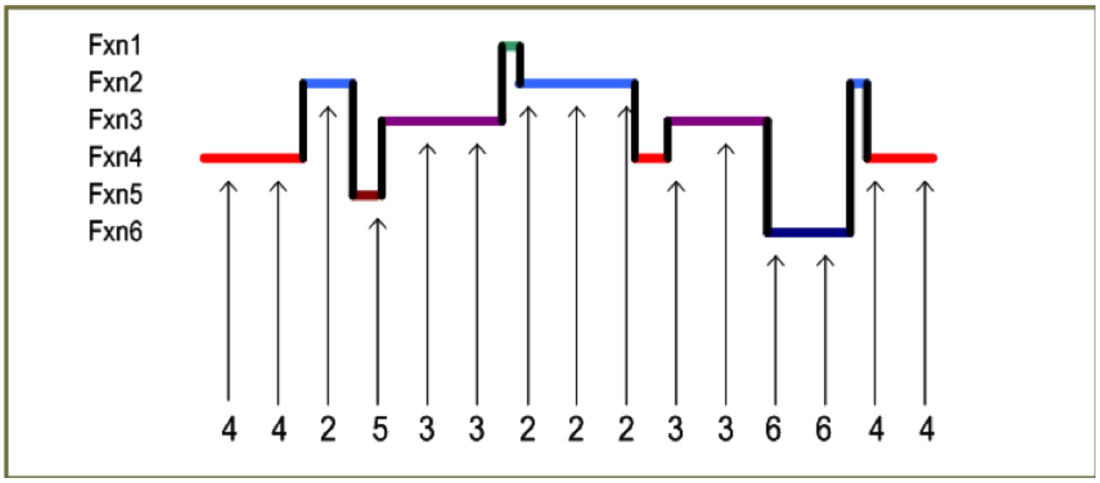


Figure 1: Statistical Profiling samples function execution periodically (Source: https://web.archive.org/web/20210114141352/http://processors.wiki.ti.com/index.php/Statistical_Profiling (https://web.archive.org/web/20210114141352/http://processors.wiki.ti.com/index.php/Statistical_Profiling))

Perf (https://perf.wiki.kernel.org/index.php/Tutorial) is a profiler tool for Linux-based systems used as an event-based profiler and to provide the statistics of the occurrences of events. However, here we use Perf to sample the states of a process and then generate a report of the sampled data.

To install Perf on an Ubuntu machine, run:

```
$ sudo apt install linux-tools-common
$ sudo apt-get install -y linux-tools-aws # run this only if you're testing on a
n AWS instance
```

Create a simple Python code (atan.py) as the target program:

```
import math
for i in range(100000):
    math.atan(i)
```

Start to sample the process:

```
$ sudo perf record python3 atan.py
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.016 MB perf.data (244 samples) ]
```

Generate the report:

```
$ sudo perf report
Samples: 219  of event 'cpu-clock', Event count (approx.): 54750000
Overhead  Command  Shared Object       Symbol
   9.13%  python3  python3.5           [.] _PyLong_Frexp
   8.22%  python3  python3.5           [.] PyEval_EvalFrameEx
   5.02%  python3  python3.5           [.] 0x000000000011701f
   4.57%  python3  libc-2.23.so        [.] 0x0000000000172974
   4.11%  python3  python3.5           [.] _PyObject_GenericGetAttrWithDict
   3.65%  python3  libm-2.23.so        [.] 0x0000000000066b85
   3.20%  python3  python3.5           [.] PyDict_GetItem
   3.20%  python3  python3.5           [.] PyFloat_AsDouble
   2.74%  python3  python3.5           [.] PyObject_Malloc
   1.83%  python3  libc-2.23.so        [.] 0x0000000000172a6e
   1.83%  python3  libm-2.23.so        [.] 0x0000000000067990
   1.83%  python3  python3.5           [.] PyDict_SetItem
```

In the example above, we can see a high overhead in _PyLong_Frexp, which is used for calculating f(x, e) = x * 2**e.

Information

Visualizing Perf Output (Optional Reading)

> "Flame graphs are a visualization of profiled software, allowing the most frequent code-paths to be identified quickly and accurately." You can visualize your perf output using FlameGraph (https://github.com/brendangregg/FlameGraph).

Instrumentation

Instrumentation in computer programming is the technique that a programmer adds code instructions into the target program to collect information. This provides information closely related to the context of the program; however, it may affect performance and lead to heisenbugs (https://en.wikipedia.org/wiki/Heisenbug).

Instrumentation can be implemented *manually*. The following is an example of adding a stopwatch in the code to learn how long a code block takes to execute.

```
import time

start = time.time()
print("CCbnb is the best!")
time_elapsed = time.time() - start
print(str(time_elapsed) + " seconds")
```

This simple program will output the time taken for running print("CCbnb is the best!"):

```
$ python3 run_instru1.py
CCbnb is the best!
5.888938903808594e-05 seconds
```

A much more sophisticated library for Python profiling is line_profiler which shows the execution time and hits per line of code.

Our simple Python code:

```
@profile
def main():
    for i in range(100):
        pass

    for j in range(200):
        pass

if __name__ == "__main__":
    main()
```

The line_profiler shows the hits and time elapsed of each line of the code:

```
$ pip3 install line_profiler
$ kernprof -l run_instru2.py
Wrote profile results to run_instru2.py.lprof

$ python3 -m line_profiler run_instru2.py.lprof
Timer unit: 1e-06 s

Total time: 0.0002 s
File: run_instru2.py
Function: main at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           @profile
     2                                           def main():
     3       101         38.0      0.4     19.0      for i in range(100):
     4       100         31.0      0.3     15.5          pass
     5
     6       201         68.0      0.3     34.0      for j in range(200):
     7       200         63.0      0.3     31.5          pass
```

In the above example, we can see that the content in the first for-loop has 100 hits and the second one has 200 hits. Take line 4 as instance, it had been hit 100 times, each time takes 0.3 microsecond, and the total time consumed is 31 microsecond. The total execution time of Line 5 contributes to 15.5% of the whole processing time.

Profiling Individual Components

Profiling Individual Components

Individual components of a cloud service should be profiled before testing the whole system in order to locate the bottlenecks of a system. We've divided the whole system into the following components: server code, system, database, and network.

Figure 2 is the current architecture of your CCbnb web service. You have one ELB that forwards all requests to two frontend instances using round-robin scheduling. In each frontend instance, there are several server threads running (see the optional reading section below). Each of the server threads does the same job where they keep pulling one of the requests to handle it. Most of the requests handled on the frontend server thread require access to the database; therefore, server threads send out requests to the backend server to retrieve or update the data stored in the database. The backend instance also runs many server threads to handle the requests from the frontend which mostly involves retrieving or updating data stored in the database.

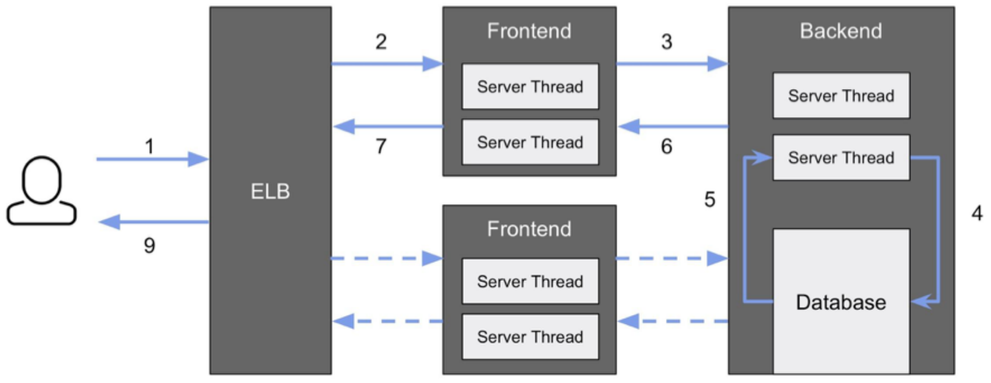


Figure 2: Overview of your CCbnb web service architecture

Please note that the CCbnb architecture doesn't represent a good architecture since it's only designed for simulating the real-world scenario. Architectures designed by students should be different from this.

Information

HTTP Connection Pool (Optional Reading)

> The process of establishing a connection from one host to another is quite complex and involves multiple packet exchanges between two endpoints, which can be quite time consuming. The overhead of connection handshaking can be significant, especially for small HTTP messages. One can achieve a much higher data throughput if open connections can be re-used to execute multiple requests. -- http://people.apache.org/~olegk/tutorial-httpclient-4.4/html/connmgmt.html (http://people.apache.org/~olegk/tutorial-httpclient-4.4/html/connmgmt.html)

HTTP Connection Pool is the idea of reusing TCP connections to receive multiple HTTP requests or to send multiple HTTP requests. This design avoids the overhead of establishing a connection and therefore it increases the throughput.

An example code from the Apache Software Foundation (http://people.apache.org/~olegk/tutorial-httpclient-4.4/html/connmgmt.html) for establishing a connection without a connection pool is shown below:

```
HttpClientContext context = HttpClientContext.create();
HttpClientConnectionManager connMrg = new BasicHttpClientConnectionManager
();
HttpRoute route = new HttpRoute(new HttpHost("localhost", 80));
// Request new connection. This can be a long process
ConnectionRequest connRequest = connMrg.requestConnection(route, null);
// Wait for connection up to 10 sec
HttpClientConnection conn = connRequest.get(10, TimeUnit.SECONDS);
try {
    // If not open
    if (!conn.isOpen()) {
        // establish connection based on its route info
        connMrg.connect(conn, route, 1000, context);
        // and mark it as route complete
        connMrg.routeComplete(conn, route, context);
    }
    // Do useful things with the connection.
} finally {
    connMrg.releaseConnection(conn, null, 1, TimeUnit.MINUTES);
}
```

If we apply a connection pool, the example code is:

```
PoolingHttpClientConnectionManager cm = new PoolingHttpClientConnectionMana
ger();
// Increase max total connection to 200
cm.setMaxTotal(200);
// Increase default max connection per route to 20
cm.setDefaultMaxPerRoute(20);
// Increase max connections for localhost:80 to 50
HttpHost localhost = new HttpHost("locahost", 80);
cm.setMaxPerRoute(new HttpRoute(localhost), 50);

CloseableHttpClient httpClient = HttpClients.custom()
        .setConnectionManager(cm)
        .build();
```

Server Code

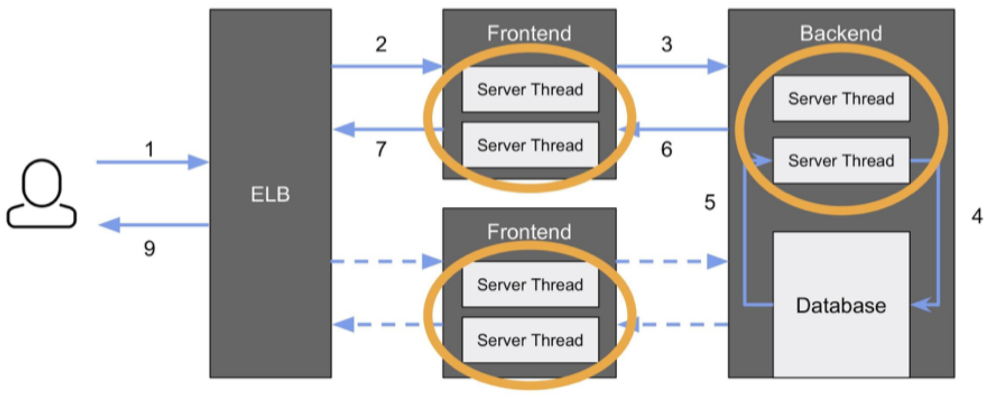


Figure 3: Server Code is running in each of your frontend/backend instance

Server code refers to the code you wrote and ran on the frontend or backend instances. It can be single threaded or multithreaded.

If your server is implemented in Python, most tools have already been covered in the above section. They are sys.setprofile (https://docs.python.org/3/library/sys.html), cProfile (https://docs.python.org/2/library/profile.html) and line_profiler (https://github.com/rkern/line_profiler).

If your server is implemented in Java, we've provided two tools for profiling in this section. We will use the same java code as an example, which is a very simple socket server that always returns “CCbnb is the best!” on port 8080.

Step 1: Copy the sample file EchoServer.java (https://s3.amazonaws.com/cmucc-public/profiling/EchoServer.java).

Step 2: Compile the source code into a class file:

```
$ javac EchoServer.java
```

Step 3: Start the server

```
$ java EchoServer
```

Step 4: Test it in another terminal session

```
$ telnet localhost 8080
Trying ::1...
Connected to localhost.
Escape character is '^]'.
CCbnb is the best!
Connection closed by foreign host.
```

Now, it’s time to try out different profiling tools!

VisualVM

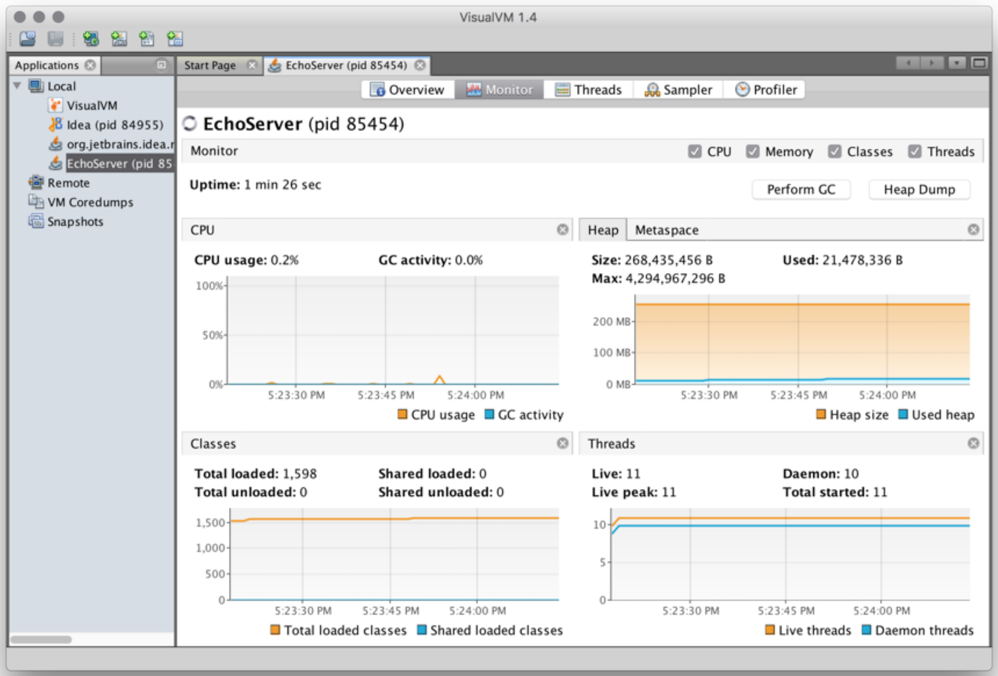
> "VisualVM is a visual tool integrating command-line JDK tools and lightweight profiling capabilities. Designed for both development and production time use."

VisualVM is a free client tool that enables you to inspect a java process including the thread states, memory usages, class loaded, etc. The Java virtual machine (JVM) has built-in instrumentation that enables you to monitor and manage it using the Java Management Extensions (JMX) technology. VisualVM takes the advantage of using Remote JMX to connect to a Java process.

Instead of running `java EchoServer` , we enable RemoteJMX by the following command:

```
$ java -Dcom.sun.management.jmxremote \
  -Dcom.sun.management.jmxremote.port=9010 \
  -Dcom.sun.management.jmxremote.local.only=false \
  -Dcom.sun.management.jmxremote.authenticate=false \
  -Dcom.sun.management.jmxremote.ssl=false \
  EchoServer
```

Download (https://visualvm.github.io/) and launch VisualVM locally, and you will be able to see the EchoServer listed on the left panel. You will also be able to see the thread states, memory usages, CPU usages, and etc. in the right window.



EchoServer (pid 85454)

Monitor

Uptime: 1 min 26 sec

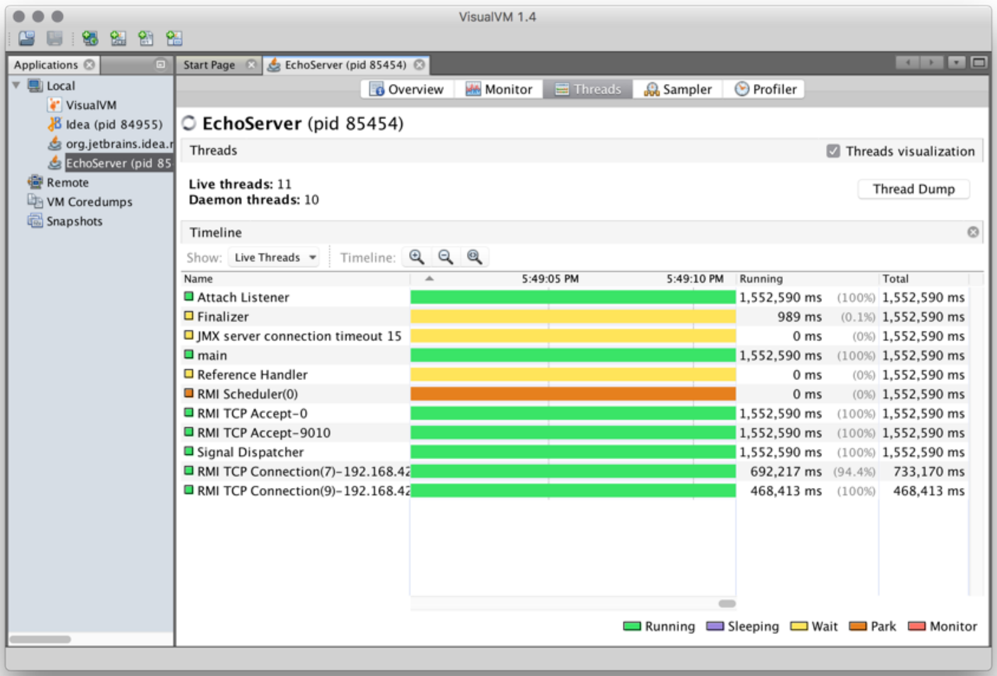
CPU usage: 0.2% GC activity: 0.0%

Size: 268,435,456 B Used: 21,478,336 B
Max: 4,294,967,296 B

Total loaded: 1,598 Shared loaded: 0
Total unloaded: 0 Shared unloaded: 0

Live: 11 Daemon: 10
Live peak: 11 Total started: 11

Figure 4: VisualVM shows the CPU and memory usage of your java process



EchoServer (pid 85454)

Threads

Live threads: 11
Daemon threads: 10

Name	5:49:05 PM 5:49:10 PM	Running		Total
🟩 Attach Listener		1,552,590 ms	(100%)	1,552,590 ms
🟨 Finalizer		989 ms	(0.1%)	1,552,590 ms
🟨 JMX server connection timeout 15		0 ms	(0%)	1,552,590 ms
🟩 main		1,552,590 ms	(100%)	1,552,590 ms
🟨 Reference Handler		0 ms	(0%)	1,552,590 ms
🟧 RMI Scheduler(0)		0 ms	(0%)	1,552,590 ms
🟩 RMI TCP Accept-0		1,552,590 ms	(100%)	1,552,590 ms
🟩 RMI TCP Accept-9010		1,552,590 ms	(100%)	1,552,590 ms
🟩 Signal Dispatcher		1,552,590 ms	(100%)	1,552,590 ms
🟩 RMI TCP Connection(7)-192.168.42		692,217 ms	(94.4%)	733,170 ms
🟩 RMI TCP Connection(9)-192.168.42		468,413 ms	(100%)	468,413 ms

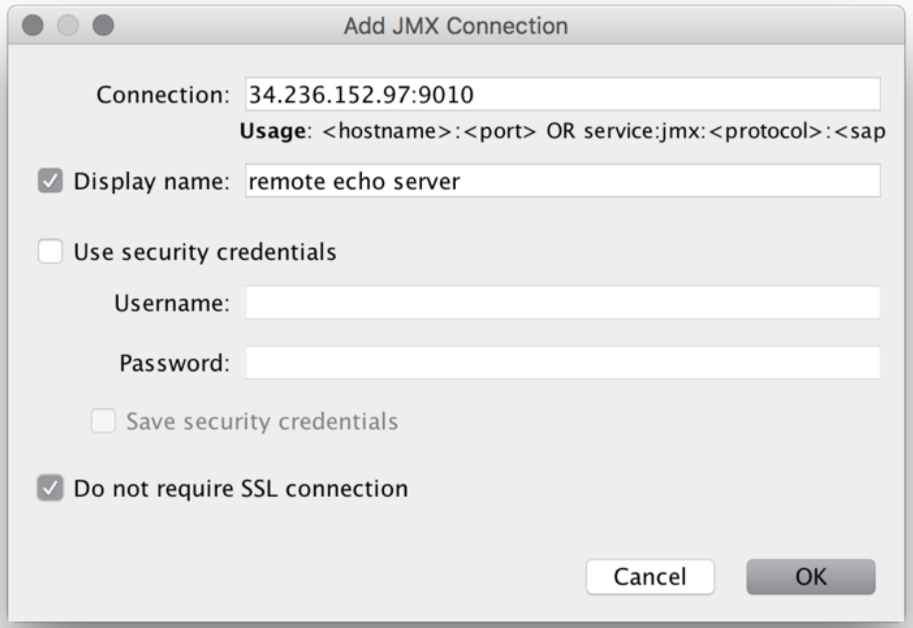
Figure 5: VisualVM shows the thread status of your java process

Memory usage information is helpful if you have implemented in-memory cache, and thread dumps is handy in case of dealing with deadlocks and race conditions.

To profile a server running on a remote machine, we need to specify the hostname while starting the server:

```
# on a remote instance
$ java -Dcom.sun.management.jmxremote \
  -Dcom.sun.management.jmxremote.port=9010 \
  -Dcom.sun.management.jmxremote.local.only=false \
  -Dcom.sun.management.jmxremote.authenticate=false \
  -Dcom.sun.management.jmxremote.ssl=false \
  -Djava.rmi.server.hostname=<YOUR_SERVER_PUBLIC_IP>
  EchoServer
```

You can connect to the remote server from your local VisualVM. Click “Add JMX Connection”, fill in your instance IP with port 9010, then check “Do not require SSL connection”. You will then be able to profile the remote server(Make sure the 9010 port of your remote server is open).



Add JMX Connection

Connection: 34.236.152.97:9010

Usage: <hostname>:<port> OR service:jmx:<protocol>:<sap

[✓] Display name: remote echo server

[ ] Use security credentials

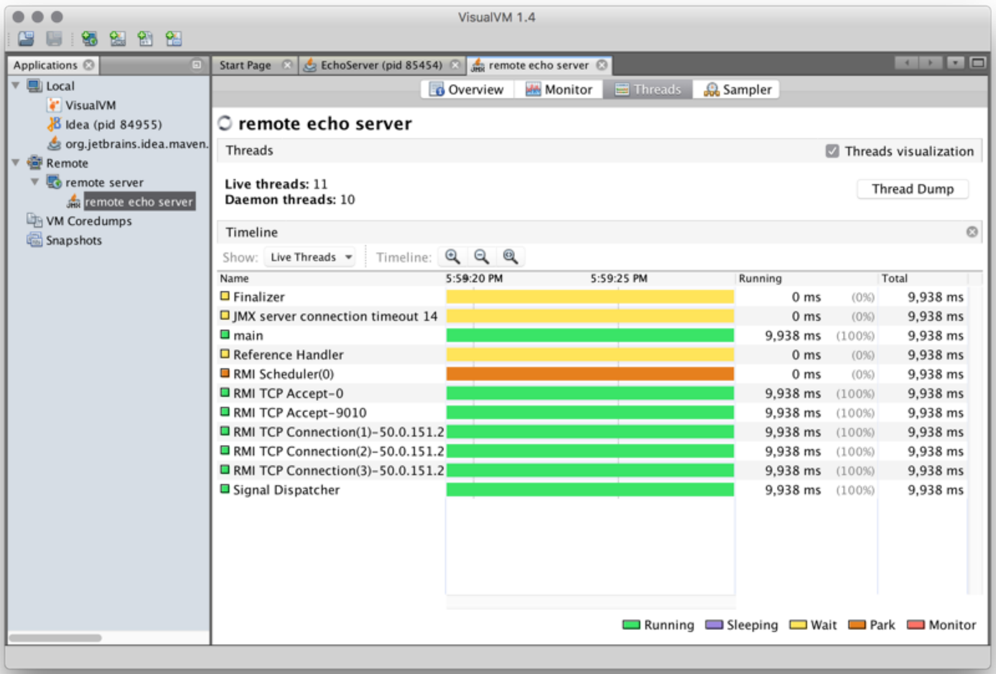
Username:

Password:

[ ] Save security credentials

[✓] Do not require SSL connection

Figure 6: Add JMX connection for connecting to a remote jvm



remote echo server

Threads

Live threads: 11
Daemon threads: 10

Name	5:59:20 PM 5:59:25 PM	Running		Total
🟨 Finalizer		0 ms	(0%)	9,938 ms
🟨 JMX server connection timeout 14		0 ms	(0%)	9,938 ms
🟩 main		9,938 ms	(100%)	9,938 ms
🟨 Reference Handler		0 ms	(0%)	9,938 ms
🟧 RMI Scheduler(0)		0 ms	(0%)	9,938 ms
🟩 RMI TCP Accept-0		9,938 ms	(100%)	9,938 ms
🟩 RMI TCP Accept-9010		9,938 ms	(100%)	9,938 ms
🟩 RMI TCP Connection(1)-50.0.151.2		9,938 ms	(100%)	9,938 ms
🟩 RMI TCP Connection(2)-50.0.151.2		9,938 ms	(100%)	9,938 ms
🟩 RMI TCP Connection(3)-50.0.151.2		9,938 ms	(100%)	9,938 ms
🟩 Signal Dispatcher		9,938 ms	(100%)	9,938 ms

🟩 Running 🟪 Sleeping 🟨 Wait 🟧 Park 🟥 Monitor

Figure 7: Monitor a remote jvm using VisualVM running on your local machine

Jvmtop

> "Jvmtop is a lightweight console application to monitor all accessible, running jvms on a machine."

To install Jvmtop, download and unzip the released code (https://github.com/patric-r/jvmtop/releases) and make sure your system had installed the SDK successfully. Then, you can execute jvmtop with one line of command:

```
$ sh ./jvmtop.sh
JvmTop 0.8.0 alpha - 21:59:33,  amd64,  4 cpus, Linux 4.4.0-109, load avg 0.00
http://code.google.com/p/jvmtop

PID MAIN-CLASS      HPCUR HPMAX NHCUR NHMAX    CPU     GC    VM USERNAME   #T DL
31806 m.jvmtop.JvmTop    52m 3463m   16m   n/a  0.00%  0.00% P8U22   ubuntu   12
31879 EchoServer         18m 3463m   11m   n/a  0.00%  0.00% P8U22   ubuntu    9
```

Jvmtop shows all your running JVMs including the memory usage, garbage collector status, etc. To retrieve more information for one of the process, you can pass the PID as a parameter. In this example, we use PID 31879, which is the PID of the socket server.

```
$ sh ./jvmtop.sh <YOUR_SERVER_PID>
JvmTop 0.8.0 alpha - 22:07:19,  amd64,  4 cpus, Linux 4.4.0-109, load avg 0.04
http://code.google.com/p/jvmtop

PID 31879: EchoServer
ARGS:
VMARGS:
VM: Private Build OpenJDK 64-Bit Server VM 1.8.0_222
UP:  0: 3m  #THR: 9    #THRPEAK: 10   #THRCREATED: 10   USER: ubuntu
GC-Time:  0: 0m   #GC-Runs: 1         #TotalLoadedClasses: 1597
CPU:  0.00% GC:  0.00% HEAP:  48m /3463m NONHEAP:  14m /  n/a

TID   NAME                                 STATE    CPU  TOTALCPU BLOCKEDBY
 12 RMI TCP Connection(2)-172.31.1       RUNNABLE  0.32%   22.47%
 15 JMX server connection timeout   TIMED_WAITING  0.13%    1.58%
 13 RMI Scheduler(0)                TIMED_WAITING  0.00%    0.01%
 11 RMI TCP Accept-0                     RUNNABLE  0.00%    0.05%
  9 Attach Listener                      RUNNABLE  0.00%    4.11%
  4 Signal Dispatcher                    RUNNABLE  0.00%    0.01%
  3 Finalizer                             WAITING  0.00%    0.01%
  2 Reference Handler                     WAITING  0.00%    0.01%
  1 main                                 RUNNABLE  0.00%    1.11%
 Note: Only top 10 threads (according cpu load) are shown!
```

System

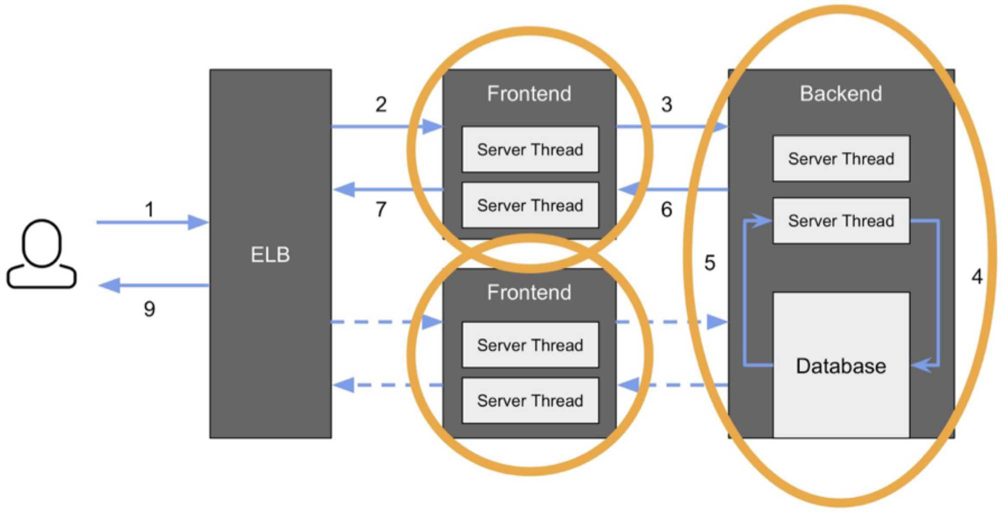
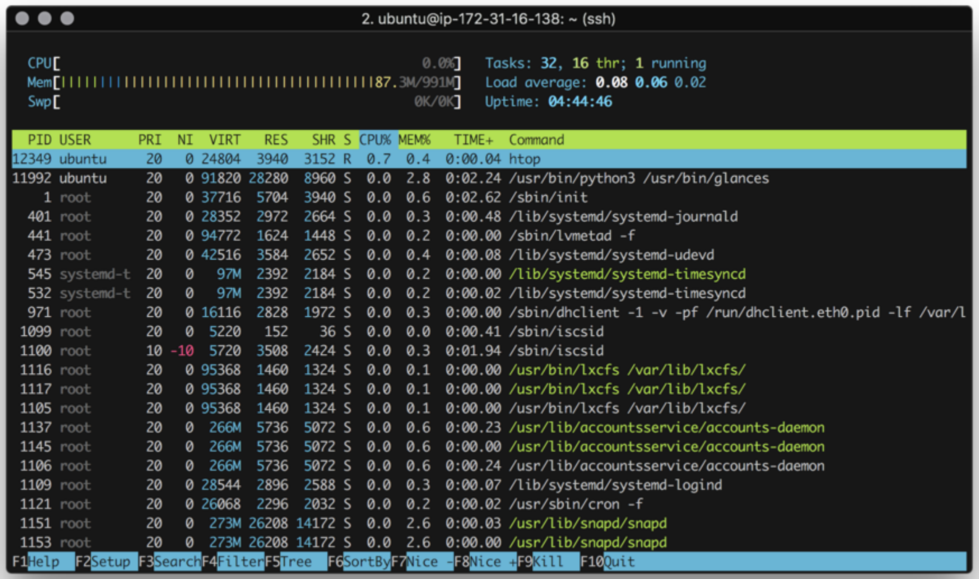


Figure 8: In each instance you allocate, there's a set of system resource for you to profile

In this part, we will focus on profiling resources on one machine in order to cover the status of the memory, CPU, disk I/O, and network. Again, the purpose of profiling each component of your system is to understand the capacity of the components, and to locate the bottleneck in your stack.

Memory & CPU

htop (https://hisham.hm/htop/): "htop is an interactive system-monitor process-viewer and process-manager. It is designed as an alternative to the Unix program top." Run `sudo apt-get install -y htop` to install on Ubuntu machines. In htop, you can view all the processes running on the instance and their memory and CPU usages. On the top left section, you can view the total memory and CPU usage.



```
2. ubuntu@ip-172-31-16-138: ~ (ssh)

  CPU[                                        0.0%]   Tasks: 32, 16 thr; 1 running
  Mem[||||||||||||||||||||||||||||||||||||||87.3M/991M]   Load average: 0.08 0.06 0.02
  Swp[                                       0K/0K]   Uptime: 04:44:46

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
12349 ubuntu     20   0 24804  3940  3152 R  0.7  0.4  0:00.04 htop
11992 ubuntu     20   0 91820 28280  8960 S  0.0  2.8  0:02.24 /usr/bin/python3 /usr/bin/glances
    1 root       20   0 37716  5704  3940 S  0.0  0.6  0:02.62 /sbin/init
  401 root       20   0 28352  2972  2664 S  0.0  0.3  0:00.48 /lib/systemd/systemd-journald
  441 root       20   0 94772  1624  1448 S  0.0  0.2  0:00.00 /sbin/lvmetad -f
  473 root       20   0 42516  3584  2652 S  0.0  0.4  0:00.08 /lib/systemd/systemd-udevd
  545 systemd-t  20   0   97M  2392  2184 S  0.0  0.2  0:00.00 /lib/systemd/systemd-timesyncd
  532 systemd-t  20   0   97M  2392  2184 S  0.0  0.2  0:00.02 /lib/systemd/systemd-timesyncd
  971 root       20   0 16116  2828  1972 S  0.0  0.3  0:00.00 /sbin/dhclient -1 -v -pf /run/dhclient.eth0.pid -lf /var/l
 1099 root       20   0  5220   152    36 S  0.0  0.0  0:00.41 /sbin/iscsid
 1100 root       10 -10  5720  3508  2424 S  0.0  0.3  0:01.94 /sbin/iscsid
 1116 root       20   0 95368  1460  1324 S  0.0  0.1  0:00.00 /usr/bin/lxcfs /var/lib/lxcfs/
 1117 root       20   0 95368  1460  1324 S  0.0  0.1  0:00.00 /usr/bin/lxcfs /var/lib/lxcfs/
 1105 root       20   0 95368  1460  1324 S  0.0  0.1  0:00.00 /usr/bin/lxcfs /var/lib/lxcfs/
 1137 root       20   0  266M  5736  5072 S  0.0  0.6  0:00.23 /usr/lib/accountsservice/accounts-daemon
 1145 root       20   0  266M  5736  5072 S  0.0  0.6  0:00.00 /usr/lib/accountsservice/accounts-daemon
 1106 root       20   0  266M  5736  5072 S  0.0  0.6  0:00.24 /usr/lib/accountsservice/accounts-daemon
 1109 root       20   0 28544  2896  2588 S  0.0  0.3  0:00.07 /lib/systemd/systemd-logind
 1121 root       20   0 26068  2296  2032 S  0.0  0.2  0:00.02 /usr/sbin/cron -f
 1151 root       20   0  273M 26208 14172 S  0.0  2.6  0:00.03 /usr/lib/snapd/snapd
 1153 root       20   0  273M 26208 14172 S  0.0  2.6  0:00.00 /usr/lib/snapd/snapd
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 9: htop

free (http://man7.org/linux/man-pages/man1/free.1.html): free is a build-in tool on Linux system that shows the memory usage.

```
$ free
              total        used        free      shared  buff/cache   available
Mem:        1014552       50620      184236        3184      779696      784924
Swap:             0           0           0
$ free -h
              total        used        free      shared  buff/cache   available
Mem:           990M         49M        179M        3.1M        761M        766M
Swap:            0B          0B          0B
```

Information

Tips

In some cases, when you run out of memory, it leads to high CPU usage. That is because, under a limited amount of memory space, the garbage collector has to clean memory much more often than usual and this consumes a lot of CPU cycles.

Different types of cloud instances contain a different number of vCPUs. To use the resource more efficiently, you should try to equally utilize all vCPUs.

For Java programs, you can adjust the heap size by giving -Xmx and -Xms parameter.

Disk

Please refer to the Storage Benchmarking primer (https://theproject.zone/s21-15619/storage-benchmarking) for tools including df, sysbench, etc.

A disk's I/O operations are relatively slow compared to other components like memory or internal network bandwidth. The following commands show that the disk only operates at around 22 MB/s for reads and about 15 MB/s for writes in a random read write test on a m4.large instance. Please try the following yourself on a cloud instance.

```
# Make sure you install the latest sysbench since there's a file-not-found bug i
n early versions
$ curl -s https://packagecloud.io/install/repositories/akopytov/sysbench/script.
deb.sh | sudo bash
$ sudo apt -y install sysbench

# Prepare data file for the test
$ mkdir /tmp/t && cd /tmp/t
$ sysbench --test=fileio --file-total-size=4G prepare

# Start the test with maximum time 60 seconds
$ sysbench --test=fileio --file-total-size=1G --file-test-mode=rndrw --max-time=
60 run
```

What did your results show? Given this insight, and while building a cloud service, we should try to decrease the number of disk operations since the disk as a device is low in the memory hierarchy (https://en.wikipedia.org/wiki/Memory_hierarchy). It is suitable for keeping large amount of data with low overhead but isn't a good option for frequent run-time accesses.

However, in the case you still need to access the data stored on disk, locality (https://en.wikipedia.org/wiki/Locality_of_reference) is another critical factor that affects speed. Namely, sequential read/write operations lead to predictable behavior, which makes it easier for the system to cache and prefetch the data you need. Sequential read/write operations decrease the disk seek time (https://en.wikipedia.org/wiki/Hard_disk_drive_performance_characteristics#Seek_time) on traditional hard drives. On the other hand, random read/write operations are slow and we should avoid them in our design. Try out the following commands and compare how the disk performs for sequential and random read operations.

```
$ sysbench --test=fileio --file-total-size=1G --file-test-mode=rndrd run
$ sysbench --test=fileio --file-total-size=1G --file-test-mode=seqrd run
```

What outputs did you get? Do you agree that the performance of sequential reads is higher than random reads?

Two additional tools for disk utilization are iostat and iotop.

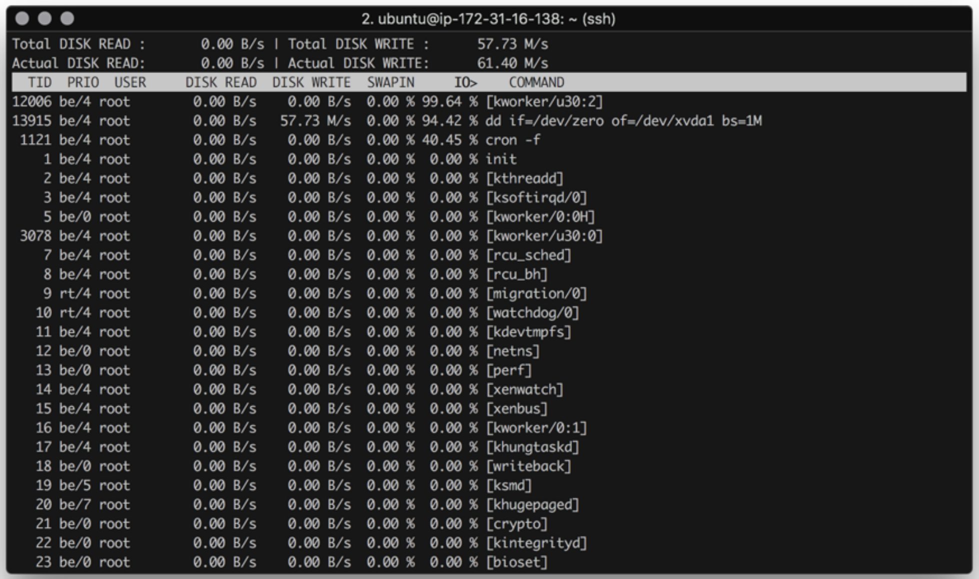
iostat (https://linux.die.net/man/1/iostat): a lightweight tool that reports I/O statistics

```
$ sudo apt install sysstat
$ iostat
    Linux 4.4.0-1094-aws (ip-172-31-1-105)  09/21/2019  _x86_64_    (4 CPU)

    avg-cpu:  %user   %nice %system %iowait  %steal   %idle
               3.97    0.00    0.95    0.02    0.00   95.06

    Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
    loop0             0.00         0.00         0.00          8          0
    nvme0n1           2.29        24.94        60.42    2236499    5417308
```

iotop (https://linux.die.net/man/1/iotop): iotop is a simple I/O monitor. Install it with sudo apt-get install -y iotop, and you will able to read the current io stat on the top of the screen. Make sure you run it with sudo.



```
2. ubuntu@ip-172-31-16-138: ~ (ssh)
Total DISK READ :       0.00 B/s | Total DISK WRITE :      57.73 M/s
Actual DISK READ:       0.00 B/s | Actual DISK WRITE:      61.40 M/s
  TID  PRIO  USER     DISK READ  DISK WRITE  SWAPIN     IO>    COMMAND
12006 be/4 root        0.00 B/s    0.00 B/s  0.00 % 99.64 % [kworker/u30:2]
13915 be/4 root        0.00 B/s   57.73 M/s  0.00 % 94.42 % dd if=/dev/zero of=/dev/xvda1 bs=1M
 1121 be/4 root        0.00 B/s    0.00 B/s  0.00 % 40.45 % cron -f
    1 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % init
    2 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kthreadd]
    3 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [ksoftirqd/0]
    5 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kworker/0:0H]
 3078 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kworker/u30:0]
    7 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcu_sched]
    8 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [rcu_bh]
    9 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [migration/0]
   10 rt/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [watchdog/0]
   11 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kdevtmpfs]
   12 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [netns]
   13 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [perf]
   14 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [xenwatch]
   15 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [xenbus]
   16 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kworker/0:1]
   17 be/4 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [khungtaskd]
   18 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [writeback]
   19 be/5 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [ksmd]
   20 be/7 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [khugepaged]
   21 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [crypto]
   22 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [kintegrityd]
   23 be/0 root        0.00 B/s    0.00 B/s  0.00 %  0.00 % [bioset]
```

Figure 10: iotop

Visualization

Visualizing the metrics discussed above is valuable in monitoring and profiling the system status. Netflix provides an open-source monitoring framework, Vector (https://github.com/Netflix/vector), which allows developers to view these metrics in the browser. Amazon CloudWatch offers a similar service; however, Vector is a free self-installed service and CloudWatch is a paid service.

Vector leverages Performance Co-Pilot (PCP), an open-source system monitoring framework, layering on top of a web UI. We first install PCP on the target system, then install Vector on the same system or our local system. Once Vector is running, we can let it connects to the system with PCP running.

In the following example, we use Ubuntu 16.04 and it's running on a t2.micro instance. First, we install PCP on this system.

```
$ curl 'https://bintray.com/user/downloadSubjectPublicKey?username=pcp' | sudo apt-key add -
$ echo "deb https://dl.bintray.com/pcp/xenial xenial main" | sudo tee -a /etc/apt/sources.list
$ sudo apt-get update
$ sudo apt-get install pcp pcp-webapi
```

Then, we start the PCP daemon:

```
$ sudo service pcp start
$ sudo service pmwebd start
```

On our local machine, we install Vector using Docker. The Docker container will be running in the background and listens to port 80.

```
$ sudo apt install docker.io
$ docker run \
  -d \
  --name vector \
  -p 80:80 \
  netflixoss/vector:latest
```

Once the container is running, we can open our browser and visit http://localhost/ (http://localhost/). After entering the DNS of the remote machine, we are now able to view the high resolution system and application metrics such as CPU utilization, network throughput, TCP connections, etc.

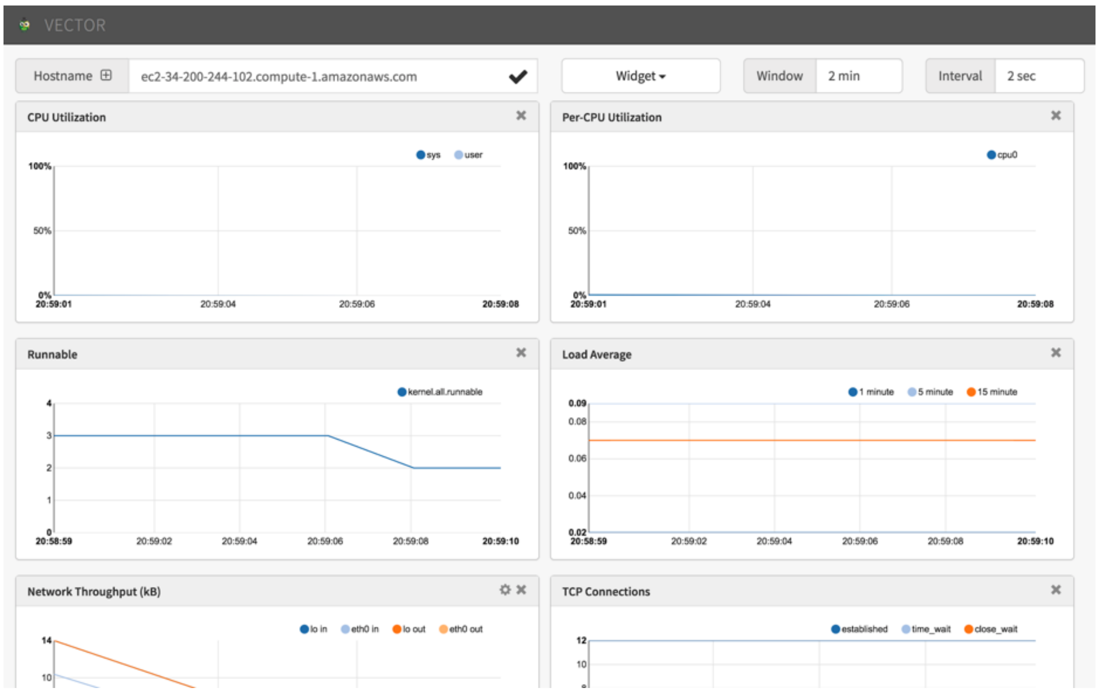

Hostname ec2-34-200-244-102.compute-1.amazonaws.com
Window 2 min
Interval 2 sec

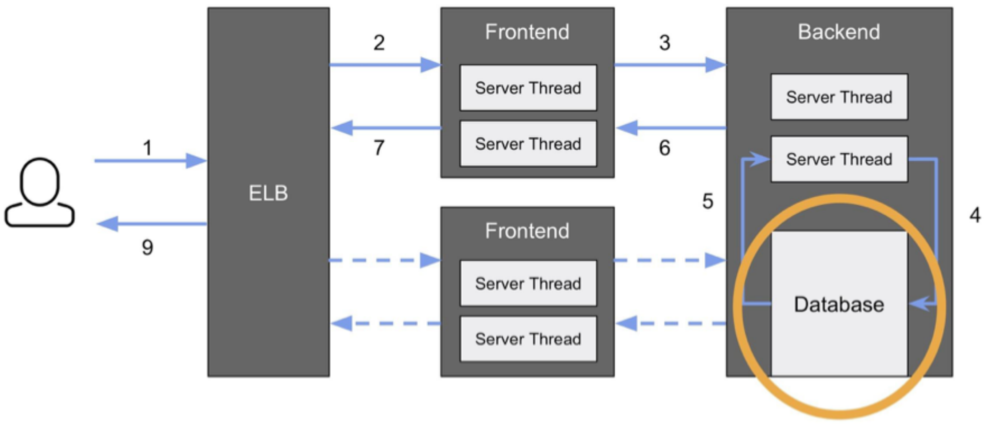
Figure 12: Screenshot of Vector

Information

Linux Performance Analysis (Optional Reading)

The article, Linux Performance Analysis in 60,000 Milliseconds (https://medium.com/netflix-techblog/linux-performance-analysis-in-60-000-milliseconds-accc10403c55), published by Netflix describes ten Linux system preloaded tools that enables developers to get a high level idea of system resource usage and running processes within 60 seconds.

Database



MySQL

While your MySQL server is receiving hundreds or thousands of queries per second which are distributed across many instances, it becomes extremely difficult to determine the locations and causes of bottlenecks without appropriate profiling techniques. In this section, we will experiment with two MySQL profiling techniques. Further, we provide you with some advanced material that will help improve your understanding. Since MySQL will be an important component in the team project, we will also introduce techniques about improving the performance of MySQL queries in this section.

Display Profiling Information

In MySQL, "The SHOW PROFILE and SHOW PROFILES statements display profiling information that indicates resource usage for statements executed during the course of the current session." (reference (https://www.digitalocean.com/community/tutorials/how-to-use-mysql-query-profiling)) In the example below, we first enable the profiling feature, then when we execute one select operation, we get the query breakdown information using SHOW PROFILES and SHOW PROFILE.

```
mysql> SET PROFILING = 1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> SELECT COUNT(*) FROM tweets WHERE tid = 447027943056113664;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)

mysql> SHOW PROFILES;
+----------+------------+----------------------------------------------------------
-----+
| Query_ID | Duration   | Query
|
+----------+------------+----------------------------------------------------------
-----+
|        1 | 0.00026200 | SELECT COUNT(*) FROM tweets WHERE tid = 44702794305611
3664 |
+----------+------------+----------------------------------------------------------
-----+
1 row in set, 1 warning (0.00 sec)

mysql> SHOW PROFILE for QUERY 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000071 |
| checking permissions | 0.000009 |
| Opening tables       | 0.000015 |
| init                 | 0.000022 |
| System lock          | 0.000009 |
| optimizing           | 0.000010 |
| statistics           | 0.000055 |
| preparing            | 0.000012 |
| executing            | 0.000003 |
| Sending data         | 0.000010 |
| end                  | 0.000003 |
| query end            | 0.000006 |
| closing tables       | 0.000007 |
| freeing items        | 0.000020 |
| cleaning up          | 0.000013 |
+----------------------+----------+
15 rows in set, 1 warning (0.00 sec)
```

Retrieve Query Execution Plan -- EXPLAIN statements

MySQL provides the `EXPLAIN` statements so that you can predict the performance of a query **without execution**. An `EXPLAIN` statement will display information from the optimizer about the statement execution plan, and you can predict performance with the information.

Important metrics output

In the table of the `EXPLAIN` result, there are some important columns which can provide insights about the performance as the following. More information can be found here (https://dev.mysql.com/doc/refman/5.7/en/explain-output.html).

1. The `rows` column indicates the number of rows MySQL predicts that it must examine to execute the query. However, The value of `rows` is a prediction and can be different from the actual rows to scan during the execution of the query. `EXPLAIN` may return different predictions of scanned rows on effectively equivalent queries. MySQL optimizer is capable to optimize most queries, but some effective equivalence is based on the specific data instead of the schema or indexes, and the optimizer cannot predict or optimize the query ahead.
2. The `Extra` column also provides valuable information. If you want to make your queries as fast as possible, watch out for Extra column values `Using filesort` and `Using temporary`. They are signals of unoptimized queries. By contrast, the following messages generally indicate the indexes are effective: `Using index`, `Select tables optimized away`.
3. The `type` can be used to indicate missing indexes or how the query should be revised. A value `const` represents that the table has only one matching row which is indexed. `const` tables are very fast because they are read only once. On the other hand, `ALL` indicates that a full table scan is needed to retrieve the record, which is very inefficient.

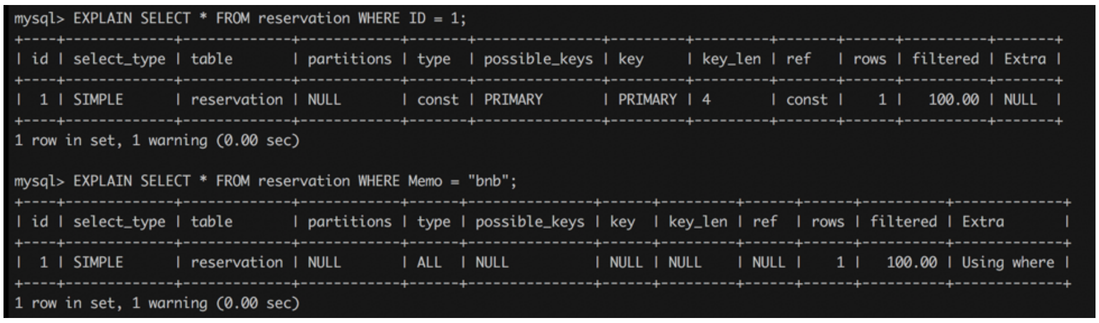
In the following example, we first create a table called reservation. The table will be used to store CCbnb reservation records so it's important to understand how well it performs while executing queries with different keys. The table we create contains three columns and only the ID column is indexed. We will introduce indexes in MySQL in more details below.

```
mysql> CREATE TABLE reservation (
    -> ID int NOT NULL AUTO_INCREMENT,
    -> Name varchar(255) NOT NULL,
    -> Memo text,
    -> PRIMARY KEY (ID)
    -> );
Query OK, 0 rows affected (0.13 sec)
```

Then, we insert a dummy row into the table:

```
mysql> INSERT INTO reservation (Name, Memo) VALUES ("CC", "bnb");
Query OK, 1 row affected (0.00 sec)
```

Now, we try to run two EXPLAIN SELECT queries with given ID and Name:



```
mysql> EXPLAIN SELECT * FROM reservation WHERE ID = 1;
+----+-------------+-------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table       | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | reservation | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+-------------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM reservation WHERE Memo = "bnb";
+----+-------------+-------------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table       | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | reservation | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    1 |   100.00 | Using where |
+----+-------------+-------------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

You can find the difference between these two queries in the `type` column which indicates the influence in the query performance of this dummy index.

Other MySQL Syntax for Profiling

```
SELECT BENCHMARK (100, 'SELECT * from t'); # Run the SELECT operation for 100 ti
mes
SHOW FULL PROCESSLIST # Show all threads and their states
SHOW ENGINE INNODB STATUS\G # Show InnoDB status
SHOW GLOBAL STATUS; # Show server runtime status
```

Tools

We've listed some tools for profiling MySQL queries below:

- General Query Log (https://dev.mysql.com/doc/refman/5.7/en/query-log.html)
- MysqlSlap (https://tosbourn.com/mysqlslap-a-quickstart-guide/)
- MySQL Workbench (https://dev.mysql.com/downloads/workbench/)

After you utilize the profiling tools to find the bottleneck for a query, the next step should be applying an optimization to address the bottleneck. We introduce some MySQL attributes below which usually affect query performance.

MySQL Indexing

Indexing is critical to query performance in MySQL. A database index helps speed up the retrieval of data from tables. When you query data from a table, MySQL checks if the indexes exist. Then MySQL uses the indexes to search the physical storage of the corresponding rows instead of scanning the whole table. Hence, you should create an index on the columns of the table from which you often query the data.

The Mechanism of Indexing

MySQL supports 2 types of indexes: B-Tree and Hash Indexes. We will explore the mechanism and the practice of B-Tree indexes.

We will use a phone book as an example to demonstrate. A CMU phone book contains all the phone numbers of every CMU member, it includes last name, first name, and phone number. The entries are sorted by last name and then first name. In MySQL syntax, this would look like the following index: `CREATE INDEX phone_index ON phone_book (last_name, first_name)`.

Schema design is based on the structure of the data; index design is based on the data as well as queries. You can build effective indexes only if you are aware of the queries you need. We will discuss different types of queries and appropriate indexes.

Index and Equality Comparison Query

If you want to search a phone number by the last name "Carnegie" and first name "Andrew". The index will narrow down the search quickly. You will look into the section where the last name starts with "C", locate "Carnegie", and search "A" to get "Andrew", and you are effectively performing a binary search.

With this phone book, you can easily find all the entries with the last name "Carnegie" without reading the whole phone book, which can be interpreted as:

```
# an index on phone_book (last_name, first_name) will help with:
SELECT * from phone_book WHERE last_name = "Carnegie";
```

However, what if you need to find all the entries with the first name "Andrew"? The only way is to scan the whole book. Anyone's first name can be "Andrew" and the distribution is not predictable with the design of the phone book.

```
# an index on phone_book (last_name, first_name) will NOT help with:
SELECT * from phone_book WHERE first_name = "Andrew";
```

To speed up the search by first name, another index is needed which groups and sorts the entries by first name, and then last name.

Index and Range Query

Similar to equality comparison queries, range queries can be faster with indexes.

Indexes will work with not only numerical columns. For example, the following query can benefit from an index on `phone_book (last_name, first_name)`:

```
SELECT * from phone_book WHERE last_name LIKE "C%";
```

Entries with the last name starting with "C" are grouped together and there is no need to scan the whole book, and `C%` is effectively a range query `WHERE last_name >= 'C' AND last_name < 'D'`.

Single-Column Index and Multiple-Column Index

For a search query that refers to multiple columns, there are two indexing approaches: create one index per column, or create one single multiple-column index (a.k.a. composite index).

```
# two single-column indexes, one on last_name and the other on first_name,
# will help with:
SELECT * from phone_book WHERE last_name = "Carnegie" AND first_name = "Andrew";
# but a composite index on (last_name, first_name) is more effective

# two single-column indexes, one on last_name and the other on first_name,
# are as good as it gets:
SELECT * from phone_book WHERE last_name = "Carnegie" OR first_name = "Andrew";
# but a composite index on (last_name, first_name) will NOT work
# because the distribution of first_name = "Andrew" is not predictable.
```

For composite indexes, **the column order matters**. If you have a query to select the rows `WHERE A > 10 and B = 10`, an index on `(B, A)` is much more effective than one on `(A, B)`.

The Mistakes of Indexing

When the workload is read-heavy, indexing is powerful to improve performance. However, you should not overuse indexing everywhere because of the following trade-offs.

Space. Each index takes extra storage space on disk to build the data structure of the index.

Slow down writes. Each index needs to be modified during writes (`INSERT` , `UPDATE` , `DELETE`).

The Auditing of Indexing

You can use `EXPLAIN` to predict your query performance as mentioned in the MySQL profiling section. If you create an index, but it cannot speed up the query, it means that either the index cannot match the need of the query, or the query is not designed properly and cannot be optimized by MySQL. We expect you to explore and experiment to find solutions when you encounter this issue before you ask on Piazza.

The Design of Indexing

Relational schema design is based on data; index design is based on queries. You should never start creating indexes without knowing the queries. You can start the design of indexes by considering the following points:

- What tables and columns do you need to query?
- What JOINS do you need to perform?
- What GROUP BY's and ORDER BY's do you need?

HBase

HBase and Yarn web-based user interfaces are the best way to profile a running HBase service. Please refer to the HBase Basics (https://theproject.zone/s21-15619/hbase) primer.

Network

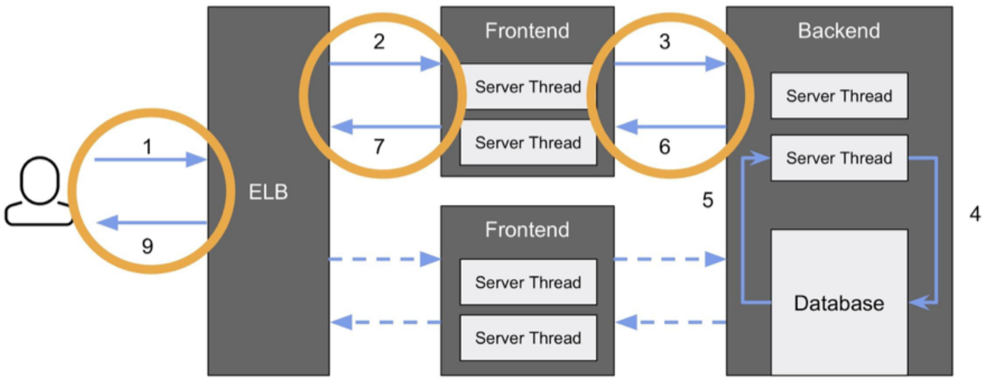


Figure 13: In the network part, we discuss the route between two instances, and their delays.

Before executing profiling experiments on your network, it's important to understand what we are measuring. The end-to-end delay (https://en.wikipedia.org/wiki/End-to-end_delay) refers to the time taken for a packet to be transmitted across a network from source to destination and is usually the delay observed on a single machine that requires data from a remote machine. The end-to-end delay can be decomposed as follows:

$$d_{\text{end-end}} = N[\, d_{\text{trans}} + d_{\text{prop}} + d_{\text{proc}} + d_{\text{queue}}]$$

where

$d_{\text{end-end}}$ = end-to-end delay

d_{trans} = transmission delay

d_{prop} = propagation delay

d_{proc} = processing delay

d_{queue} = Queuing delay

- Transmission delay: the time required to push all the packet's bits into the wire
- Propagation delay: the time it takes one bit to travel from one end of the "wire" to the other
- Processing delay: the time it takes routers to process the packet header, which is the key component of network delay
- Queuing delay: the time a job waits in a queue until it can be executed

Find the route packets take between two machines

We will use traceroute to print the route between two machines. Traceroute (https://en.wikipedia.org/wiki/Traceroute) is a computer network diagnostic tool for displaying the route and measuring transit delays of packets across an Internet Protocol network.

To install traceroute, run following command in your AWS instance:

```
$ sudo apt-get install inetutils-traceroute
```

To find out the route between your instance the cmu.edu web server, run:

```
$ traceroute cmu.edu
traceroute to cmu.edu (128.2.42.10), 64 hops max
  1   216.182.226.162  16.215ms  24.775ms  21.919ms
  2   100.66.8.84  15.812ms  22.176ms  21.750ms
  3   100.66.10.166  15.329ms  21.912ms  21.971ms
  4   100.66.7.45  17.932ms  21.955ms  21.959ms
  5   100.66.4.225  15.700ms  21.692ms  21.962ms
  6   100.65.10.97  0.305ms  0.287ms  0.290ms
  7   52.93.25.144  2.004ms  2.043ms  2.053ms
  8   52.93.25.157  0.909ms  0.934ms  0.893ms
  9   54.239.108.124  5.320ms  4.760ms  5.869ms
 10   54.239.108.103  1.013ms  1.336ms  1.409ms
 11   162.252.69.214  0.828ms  0.745ms  0.748ms
 12   162.252.70.74  1.108ms  1.038ms  1.196ms
 13   162.252.70.136  1.686ms  1.717ms  1.552ms
 14   64.57.20.138  17.024ms  6.600ms  6.495ms
 15   204.238.76.234  6.440ms  6.365ms  6.448ms
 16   204.238.76.50  22.432ms  24.370ms  22.383ms
 17   *  *  *
 18   128.2.255.249  26.200ms  26.206ms  26.218ms
 19   128.2.255.210  27.593ms  38.952ms  27.606ms
 20   128.2.42.10  26.425ms  26.428ms  26.453ms
```

As shown in the output, there are 20 hops between our AWS instance the cmu.edu web server. Traceroute sends out three packets per hop, and each column corresponds to the time it takes to get one of the packages back (round-trip delay time, RTT).

However, if we try to find out the route between two AWS instances located in the same subnet, the number of hops is 1;

```
$ traceroute 172.31.26.175
traceroute to 172.31.26.175 (172.31.26.175), 64 hops max
  1   172.31.26.175  1.019ms  0.904ms  0.916ms
```

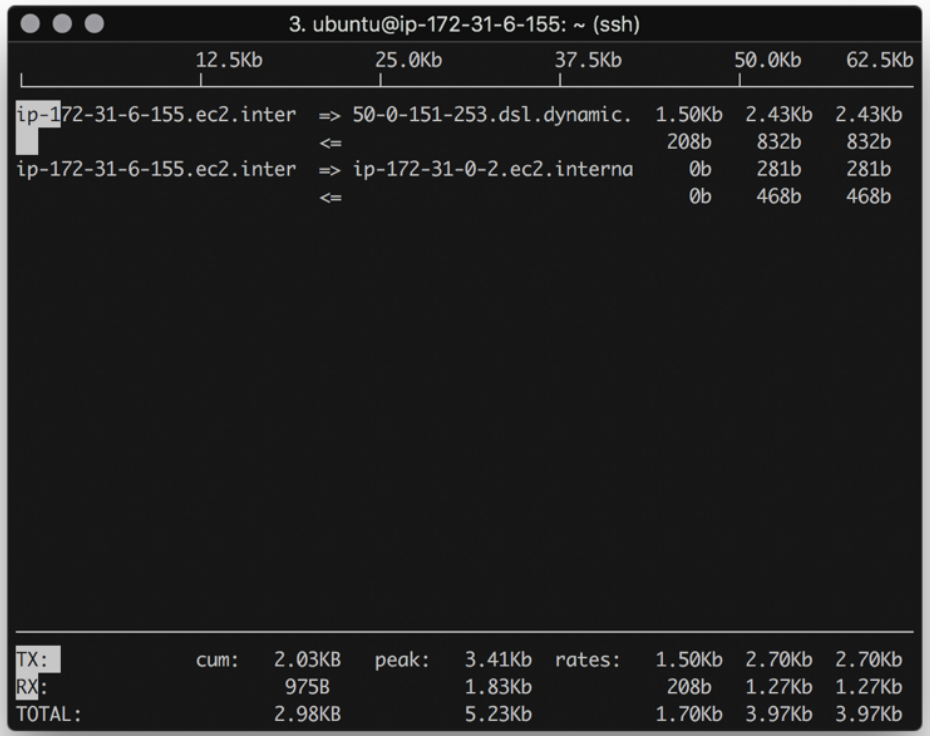
Measure the delay between two machines

Although you can retrieve the delay information using traceroute, there’s an easier way to get an end-to-end delay between two instances with ping. In the following example, we found that the delay between AWS instance the cmu.edu web server is around 26 ms, and the delay between two AWS instances is only about 1 ms.

```
$ ping ip-172-31-26-175.ec2.internal
PING ip-172-31-26-175.ec2.internal (172.31.26.175) 56(84) bytes of data.
64 bytes from ip-172-31-26-175.ec2.internal (172.31.26.175): icmp_seq=1 ttl=64 t
ime=1.03 ms
64 bytes from ip-172-31-26-175.ec2.internal (172.31.26.175): icmp_seq=2 ttl=64 t
ime=1.01 ms
--- ip-172-31-26-175.ec2.internal ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 1.017/1.025/1.034/0.033 ms
ubuntu@ip-172-31-6-155:~$ ping cmu.edu
PING cmu.edu (128.2.42.10) 56(84) bytes of data.
64 bytes from CMU-VIP.ANDREW.CMU.EDU (128.2.42.10): icmp_seq=1 ttl=233 time=26.4
ms
64 bytes from CMU-VIP.ANDREW.CMU.EDU (128.2.42.10): icmp_seq=2 ttl=233 time=26.4
ms
--- cmu.edu ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 26.430/26.450/26.470/0.020 ms
```

Measure the bandwidth usage on an interface

iftop (http://www.ex-parrot.com/pdw/iftop/): display bandwidth usage on an interface by host



```
3. ubuntu@ip-172-31-6-155: ~ (ssh)
                   12.5Kb        25.0Kb        37.5Kb        50.0Kb     62.5Kb

ip-172-31-6-155.ec2.inter  => 50-0-151-253.dsl.dynamic.  1.50Kb  2.43Kb  2.43Kb
                           <=                             208b    832b    832b
ip-172-31-6-155.ec2.inter  => ip-172-31-0-2.ec2.interna     0b    281b    281b
                           <=                               0b    468b    468b

TX:             cum:   2.03KB   peak:   3.41Kb  rates:   1.50Kb  2.70Kb  2.70Kb
RX:                     975B            1.83Kb            208b   1.27Kb  1.27Kb
TOTAL:                 2.98KB           5.23Kb           1.70Kb  3.97Kb  3.97Kb
```

Figure 14: Screenshot of iftop

nstat (https://loicpefferkorn.net/2016/03/linux-network-metrics-why-you-should-use-nstat-instead-of-netstat/): network statistics tools

```
$ netstat --statistics
Ip:
    6870 total packets received
    4 with invalid addresses
...
IpExt:
    InOctets: 9763516
    OutOctets: 1415322
    InNoECTPkts: 12584
```

Integration

Integration

In this section, you should have profiled each component of your system, and are ready to profile the full system. Building your own load generator will allow you to see how well your system performs, and provides you with details of the client states. You will be more confident in what your clients will experience when they use your service. Hence, profiling your entire system with a load generator is an important skill to learn.

Load Generator

Before using a sophisticated load generating tool, it's worthwhile to test your system with a lightweight HTTP request client. This allows you to have more control on testing specific datasets you generated, and to test out corner cases before launching a massive load test with uncertainties. *Remember testing and profiling should be done incrementally.* That is, if you skip the middle steps, it increases the difficulties in finding a bug later.

Unirest (http://unirest.io/java.html)

"Unirest is a set of lightweight HTTP libraries available in multiple languages." Here, we provide template code that contains one load generator using Unirest and one simple server using NanoHTTPD (https://github.com/NanoHttpd/nanohttpd). The server reads input parameter n and calculates the corresponding Fibonacci number (https://en.wikipedia.org/wiki/Fibonacci_number). The load generator reads the testing data from a file (data.txt), sends out requests, and verifies the correctness.

Step 1. Download and build the sample project

```
$ wget https://s3.amazonaws.com/cmucc-public/profiling/unirest.tar.gz
$ tar -xvzf unirest.tar.gz --one-top-level
$ cd unirest
$ mvn clean package
```

Step 2: Run server

```
$ mvn exec:java -Dexec.mainClass=Server
```

Step 3: Run load generator

```
$ mvn exec:java -Dexec.mainClass=LoadGen
```

In LoadGen.java, we use a manual instrumentation technique to print out the time taken for testing the whole dataset (in our case, it's around 12.5 seconds).

```
Done, time took: 12574 ms
```

Then, let's edit the server code to make the server compute the Fibonacci number faster. Try to find out the inefficient recursion function and replace it with a better implementation. Then, run the test again:

```
$ mvn exec:java -Dexec.mainClass=Server
$ mvn exec:java -Dexec.mainClass=LoadGen
```

Now, it should take a shorter time to complete the test (around 1.3 seconds).

You are free to experiment by adding more features on top of the load generator to test your own server. You may want to create multiple threads in the load generator to see how your server reacts.

Finally, it's time for you to test your system with a large scale of dataset. JMeter and Apache HTTP server benchmarking tool (AB) are two handy load generators which allow you to create customized test plans. Instead of providing detail instructions, we leave this part for you to explore.

- JMeter (http://jmeter.apache.org/)
- Apache HTTP server benchmarking tool (AB) (https://httpd.apache.org/docs/2.4/programs/ab.html)

Advanced Topics

Logging

"Logging is the process of writing log messages during the execution of a program to a central location. This logging allows you to report and persist error and warning messages as well as information messages (e.g., runtime statistics) so that the messages can later be retrieved and analyzed." - http://www.vogella.com/tutorials/Logging/article.html (http://www.vogella.com/tutorials/Logging/article.html)

While debuggers aren't available in multithreaded applications or distributed applications, logging became the only way for us to debug a complex service. Experience indicates that logging is an important factor in the development cycle because it offers precise context of the processes and the log outputs can be stored in persistent disk to be studied later.

However, logging is not the main focus of most programs we build, a simple to understand and to use library helps us gain all the benefits of logging without spending too much effort. We will introduce *log4j*, a common logging library for Java, and provide example code on how it can be used.

First, add log4j dependency into your pom.xml file:

```
<dependencies>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>
</dependencies>
```

Retrieve one logger instance for your class:

```
private static final Logger logger = Logger.getLogger(<classname>.class);
```

Then, setup your logger level. In our case, we set to DEBUG level for all loggers under this execution.

```
BasicConfigurator.configure();
Logger.getRootLogger().setLevel(Level.FATAL);
```

Nice, and it's time for us to print out some logs:

```
logger.debug("I am the best developer at CCbnb!");
```

Printing too many debug messages will slow down your processes; therefore, instead of commenting the debug states, it's much more convenient if we set different logger levels for debug and for deployment. That is, you can set logger level to ERROR in a deployed server and loggers will not log messages at the DEBUG level. The following code is an example, and you can try to change the logger levels.

```
import org.apache.log4j.BasicConfigurator;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

public class LoggingTest {

    private static final Logger logger = Logger.getLogger(LoggingTest.class);

    public static void main(String[] ars) {

        BasicConfigurator.configure();
        Logger.getRootLogger().setLevel(Level.DEBUG);

        logger.trace("trace");
        logger.debug("debug");
        logger.info("info");
        logger.warn("warn");
        logger.error("error");
        logger.fatal("fatal");

    }

}
```

In the case you don't want to recompile the Java code for different logging configuration, you can place a configuration file for log4j under src/main/resources called log4j.properties. For more information, please checkout some examples at https://www.mkyong.com/logging/log4j-log4j-properties-examples/ (https://www.mkyong.com/logging/log4j-log4j-properties-examples/).

Queueing System

To build a server with high throughput, we strive to fully utilize every resource we have. Think of your server as a queueing system, where you have incoming requests waiting for your server to process. Multiple metrics matter in this system and will affect the final throughput: they are arrival rate, service time, system capacity, number of servers, etc. The strategy that your server applies for deciding the order of which request to handle affects the fairness and the final throughput. To learn more on this topic, please read more about the Queueing theory (https://en.wikipedia.org/wiki/Queueing_theory).

References

- Bukh, Per Nikolaj D. "The art of computer systems performance analysis, techniques for experimental design, measurement, simulation and modeling." (1992): 113-115.
- ETH Advanced Systems Lab Tutorial IV Queuing Systems https://www.systems.ethz.ch/sites/default/files/file/asl2017/slides/ASL-17-Queuing-Theory.pdf
- https://www.sitepoint.com/using-explain-to-write-better-mysql-queries/ (https://www.sitepoint.com/using-explain-to-write-better-mysql-queries/)
- https://www.azul.com/blog/ (https://www.azul.com/blog/)
- http://www.brendangregg.com/flamegraphs.html (http://www.brendangregg.com/flamegraphs.html)