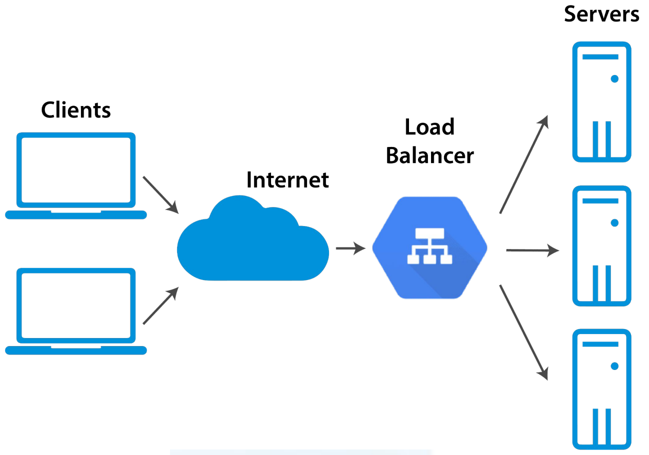
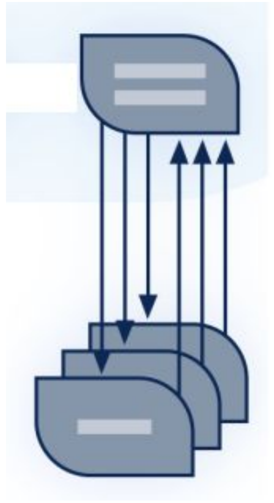
Stress/Load Testing Tool

For Web Systems

Introduction

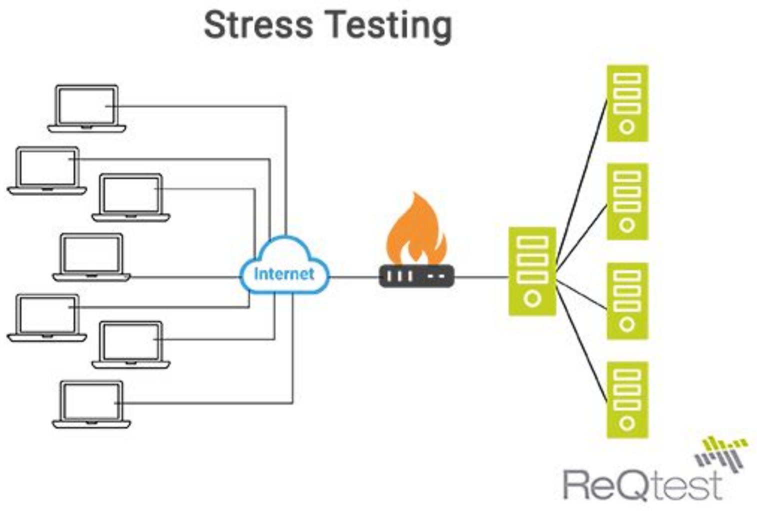
- Modern web services are designed to be horizontally scalable (through replication) to handle large scale user load
- To maintain quality of service each individual server should remain healthy in the case of infinite load on the system.
- Big companies like amazon / google to maintain their quality of service (Lesser Failure rate and send the response with in a threshold), they profile each service to determine the maximum load one individual server can handle to setup m serves to handle user traffic.





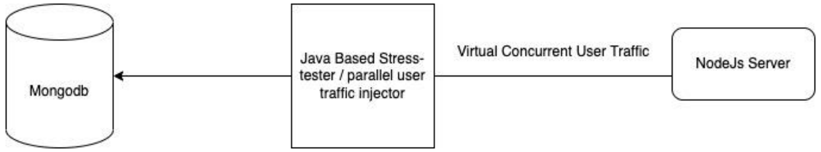
Stress Testing

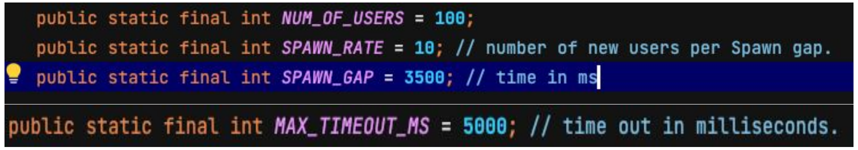
- As mentioned in the introduction to meet highest standards of quality of service, each individual service needs to profiled by injecting virtual user traffic.
- Based on the failure rate, we could determine the highest number of requests the service can handle concurrently.
- Standard tools (Examples)
    - https://jmeter.apache.org/
    - https://locust.io/



Architecture / High Level Design

- A Java based stress-testing tool / virtual traffic generator has been built as the part of the project.
- MongoDb has been used in the backend to store the data for each run / experiment that can be used to generate statistics about the server behaviour
- For the sake of experimentation a NodeJs server / service has been used.
- Virtual traffic is added incrementally to observe the server behaviour at different stages of the traffic uniquely


Mongodb
Java Based Stress-tester / parallel user traffic injector
Virtual Concurrent User Traffic
NodeJs Server



```
public static final int NUM_OF_USERS = 100;
public static final int SPAWN_RATE = 10; // number of new users per Spawn gap.
public static final int SPAWN_GAP = 3500; // time in ms
public static final int MAX_TIMEOUT_MS = 5000; // time out in milliseconds.
```

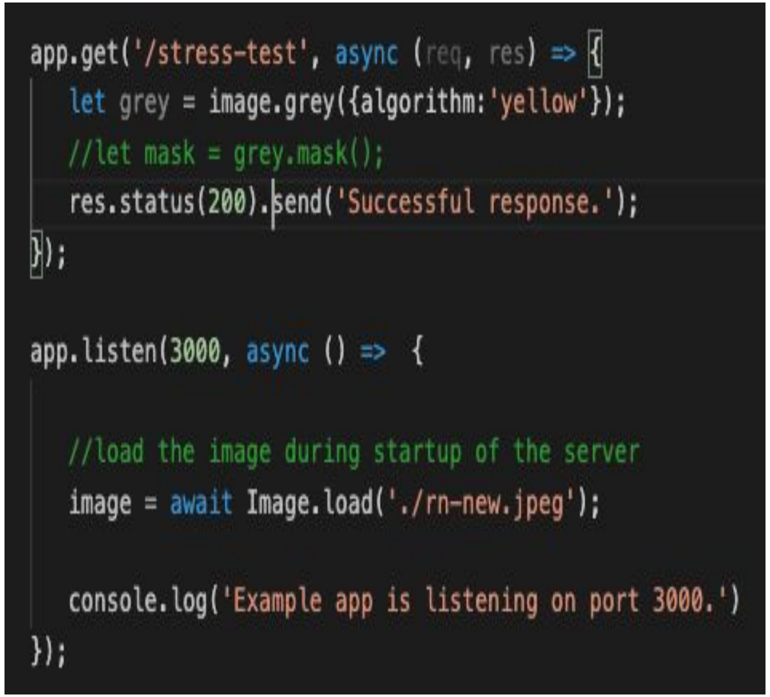
To test the maximum capacity of the load the load testing tool has been made configurable on the the following params

- NUMBER_OF_USERS (maximum number of concurrent users to be generated for the experiment)
- SPAWN_RATE (Rate at which new user / threads will be added in each interval)
- SPAWN_GAP (The Gap / Interval between new users getting added)
- MAX_TIMEOUT_MS (The maximum acceptable latency of the web server request)

The instructions for editing these parameters is mentioned in readme.md file

Node Js Server

- The server / service has a simple functionality in the stack. While the server is starting up it loads an image into the heap / memory.
- Upon each service request it tries to create a grey image (from rgb image) which is a matrix operation / CPU work.
- Node Js by design is single threaded and philosophy is good for Asynchronous IO / Network Operation and poorly performs if the services are CPU heavy
- As Node Js service is not Computation friendly we have chosen to do a matrix operations during each service request to better observe proof of concept for stress-testing tool.



```
app.get('/stress-test', async (req, res) => {
    let grey = image.grey({algorithm:'yellow'});
    //let mask = grey.mask();
    res.status(200).send('Successful response.');
});

app.listen(3000, async () =>  {

    //load the image during startup of the server
    image = await Image.load('./rn-new.jpeg');

    console.log('Example app is listening on port 3000.')
});
```

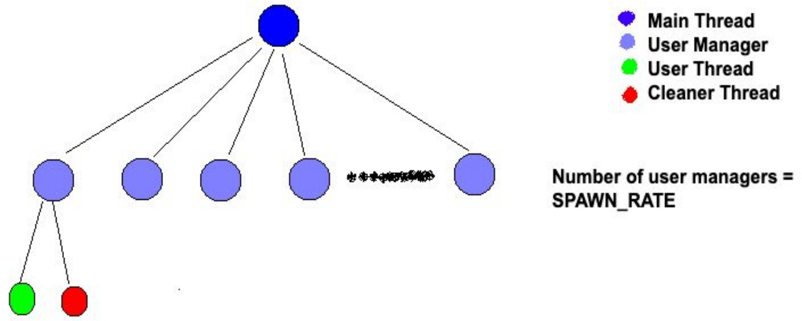
Stress tester / Virtual traffic generator

- The stress tester / virtual traffic generator tool has been built using java (can be ~multithreading based traffic generator)
- The tool generates virtual traffic in a synchronized way, adding new user threads and cleaner threads at regular intervals
- There are mainly three categories of threads that will be generated in the lifecycle of a stress-test
    - User Manager Threads
    - User Thread
    - Cleaner Thread

Details regarding each of threads and their functionality is explained in further slides.

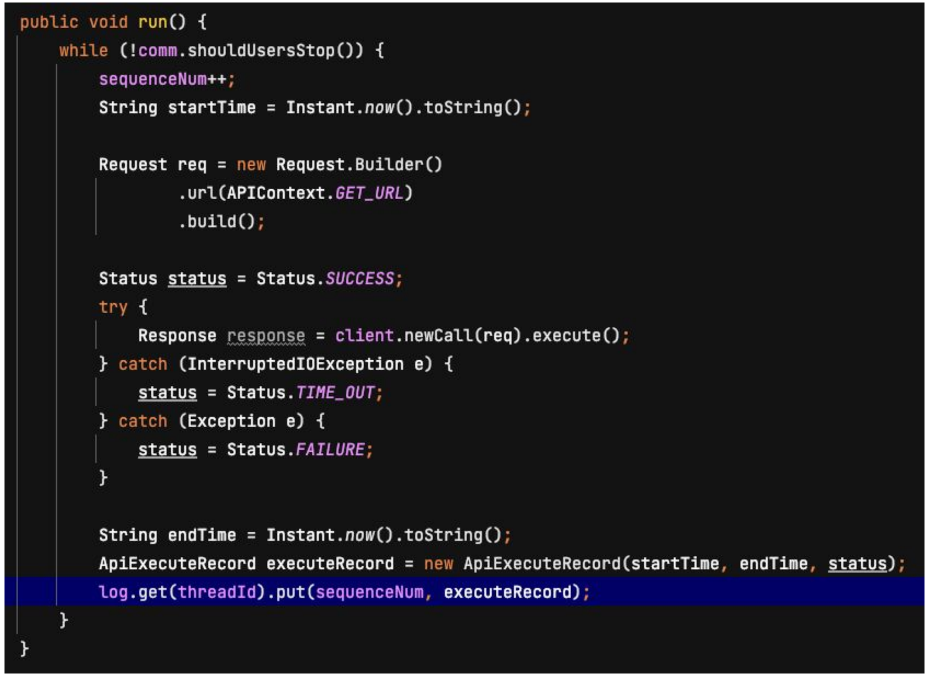
User Manager

- The main thread will create user managers who are responsible for adding new virtual user threads in a synchronized way
- Number of user manager threads = Spwan_rate.
- On a regular interval / gap of spawn_gap each user manager will add a new user thread and corresponding cleaner thread (to clean the heap data and push to mongodb)



User Thread

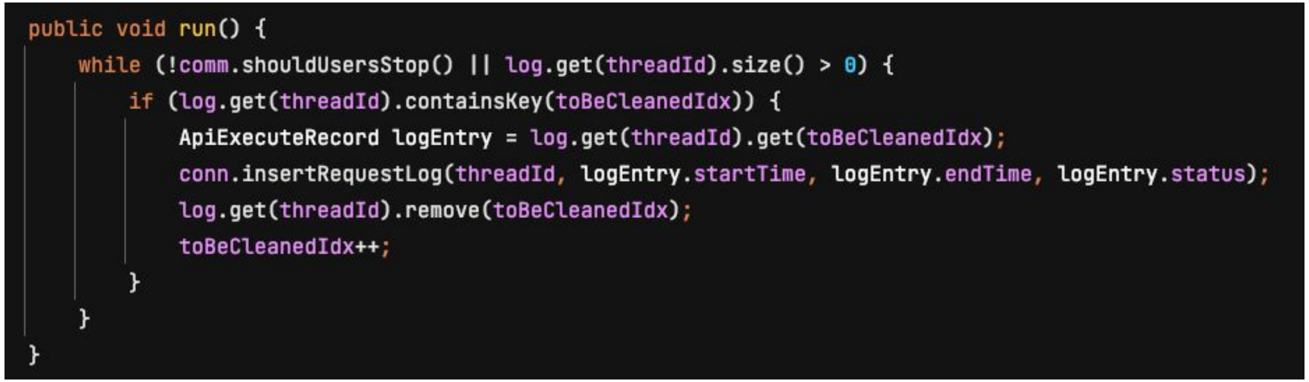
- Every User Thread has a simple task of generating traffic i.e. send a request to the web server and wait to see the response in the acceptable time
- If the response comes back in acceptable time mark the request was successful for the requestSeqNum
- Else mark it as Timed out
- The user thread does one request at a time in a synchronous way
- After each request collect the startTime, endTime and status of the request and add them to a shared data structure which will be later cleaned from memory



```java
public void run() {
    while (!comm.shouldUsersStop()) {
        sequenceNum++;
        String startTime = Instant.now().toString();

        Request req = new Request.Builder()
                .url(APIContext.GET_URL)
                .build();

        Status status = Status.SUCCESS;
        try {
            Response response = client.newCall(req).execute();
        } catch (InterruptedIOException e) {
            status = Status.TIME_OUT;
        } catch (Exception e) {
            status = Status.FAILURE;
        }

        String endTime = Instant.now().toString();
        ApiExecuteRecord executeRecord = new ApiExecuteRecord(startTime, endTime, status);
        log.get(threadId).put(sequenceNum, executeRecord);
    }
}
```

Cleaner Thread

- There is one cleaner thread created for each corresponding user thread.
- Cleaner Thread job is to garbage collect the memory that was generated by the user thread and push the statistics of requests data to mongodb
- This cleaner thread helps to keep the stress-tester process lightweight in terms of memory and since all the stats are being pushed to mongodb different statistics could be produced later.

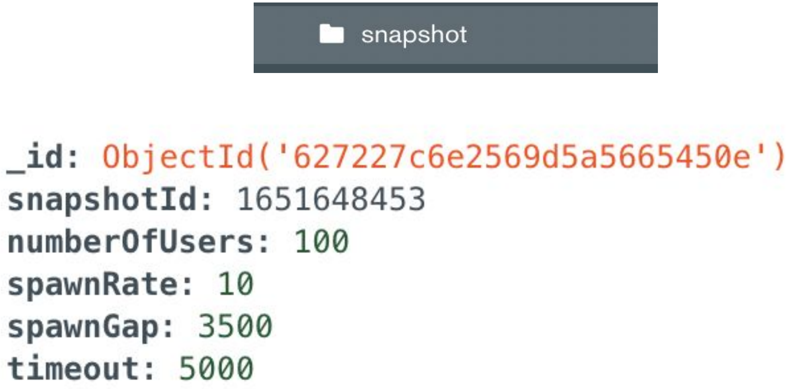


```java
public void run() {
    while (!comm.shouldUsersStop() || log.get(threadId).size() > 0) {
        if (log.get(threadId).containsKey(toBeCleanedIdx)) {
            ApiExecuteRecord logEntry = log.get(threadId).get(toBeCleanedIdx);
            conn.insertRequestLog(threadId, logEntry.startTime, logEntry.endTime, logEntry.status);
            log.get(threadId).remove(toBeCleanedIdx);
            toBeCleanedIdx++;
        }
    }
}
```

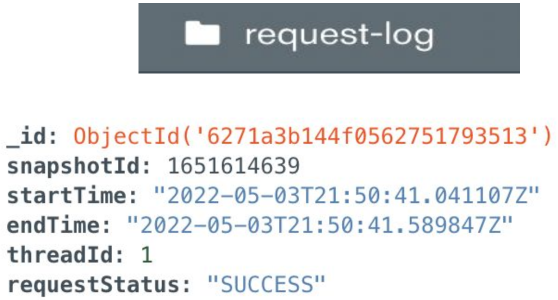
MongoDb Data

The following data points are stored in mongodb regarding each run (stress-test is the database name)

- Snapshot (Each stress-testing run is treated as one snapshot) and the details of this run is stored in the snapshot tabe / collection in mongodb
- The stats generated by user thread are cleaned by cleaner thread and added to the request-log table/collection (example document is shown in image to the side)



```
_id: ObjectId('627227c6e2569d5a5665450e')
snapshotId: 1651648453
numberOfUsers: 100
spawnRate: 10
spawnGap: 3500
timeout: 5000
```

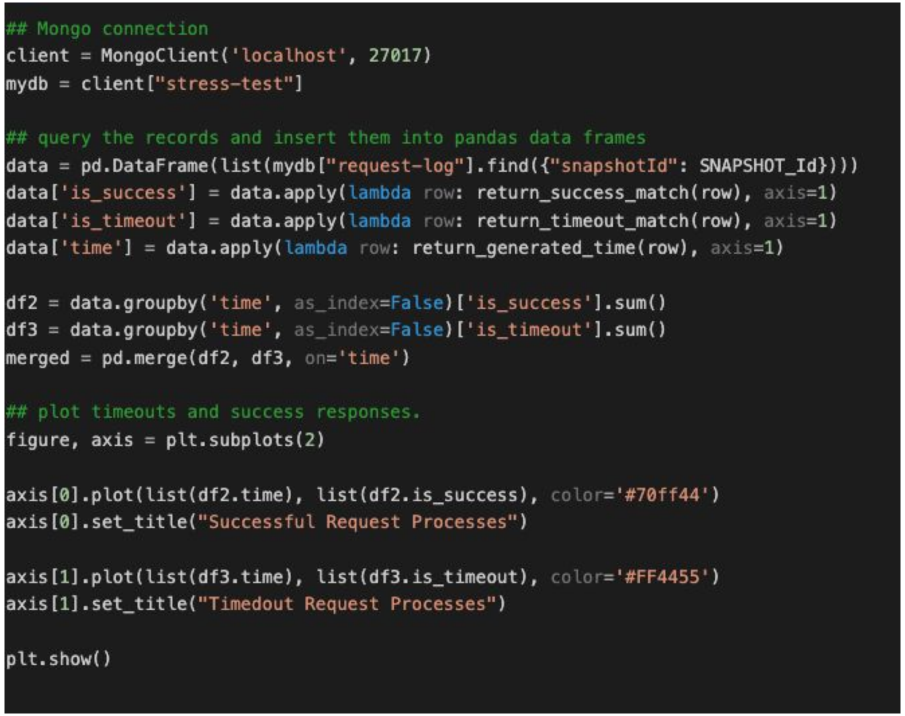


```
_id: ObjectId('6271a3b144f0562751793513')
snapshotId: 1651614639
startTime: "2022-05-03T21:50:41.041107Z"
endTime: "2022-05-03T21:50:41.589847Z"
threadId: 1
requestStatus: "SUCCESS"
```

Throughput Visualizer

A python utility has been developed that helps to generated the result graphs for each run.

The instructions for generating graph is mentioned in readme.md file

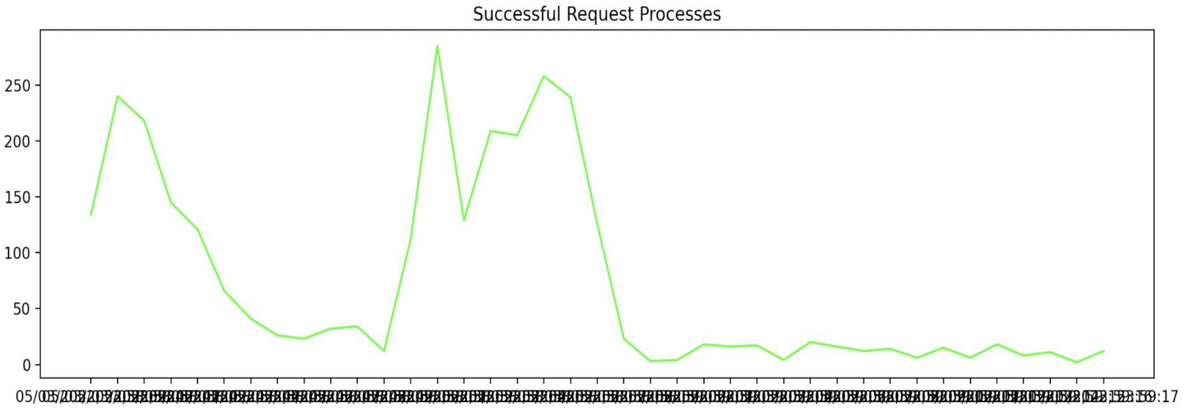


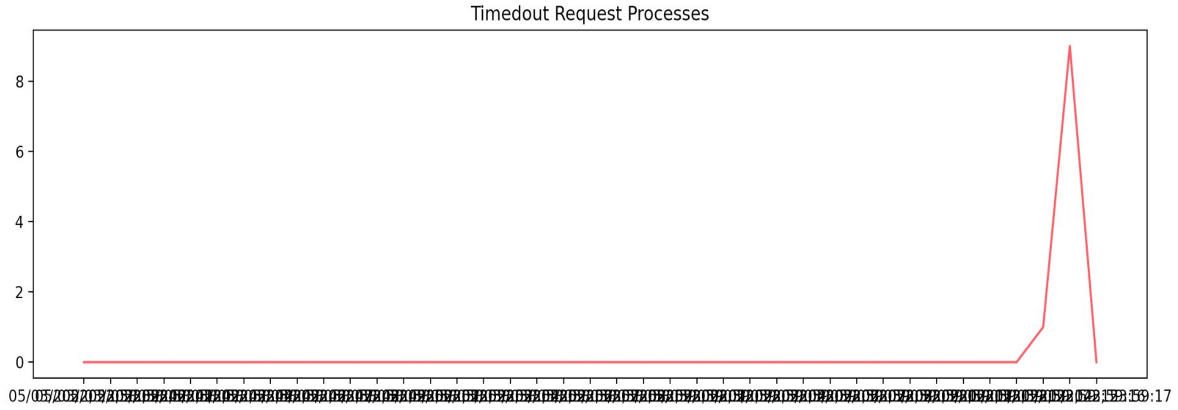
```
## Mongo connection
client = MongoClient('localhost', 27017)
mydb = client["stress-test"]

## query the records and insert them into pandas data frames
data = pd.DataFrame(list(mydb["request-log"].find({"snapshotId": SNAPSHOT_Id})))
data['is_success'] = data.apply(lambda row: return_success_match(row), axis=1)
data['is_timeout'] = data.apply(lambda row: return_timeout_match(row), axis=1)
data['time'] = data.apply(lambda row: return_generated_time(row), axis=1)

df2 = data.groupby('time', as_index=False)['is_success'].sum()
df3 = data.groupby('time', as_index=False)['is_timeout'].sum()
merged = pd.merge(df2, df3, on='time')

## plot timeouts and success responses.
figure, axis = plt.subplots(2)

axis[0].plot(list(df2.time), list(df2.is_success), color='#70ff44')
axis[0].set_title("Successful Request Processes")

axis[1].plot(list(df3.time), list(df3.is_timeout), color='#FF4455')
axis[1].set_title("Timedout Request Processes")

plt.show()
```

Results

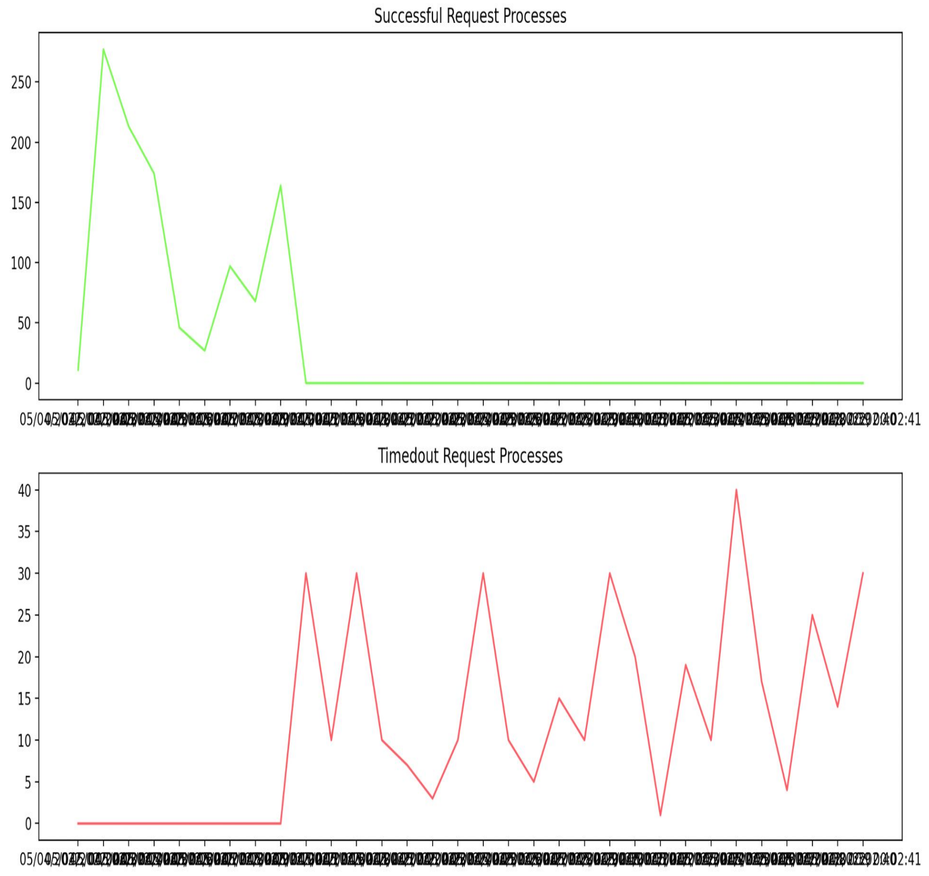
- Different Behaviours of servers under different constraints is as follows.
- The Graph in the coming slides has two graphs, the upper green graph shows throughput of requests (per second)
- The Lower Graph (in red) shows the number of timeout requests per second
- The General framework can be used to generate few more stats (per user stats, failures rates, throughput aggregated for time interval)

The inflection point (the point where server is no longer able to serve the requests) occurs much later when the allowed time out is higher.



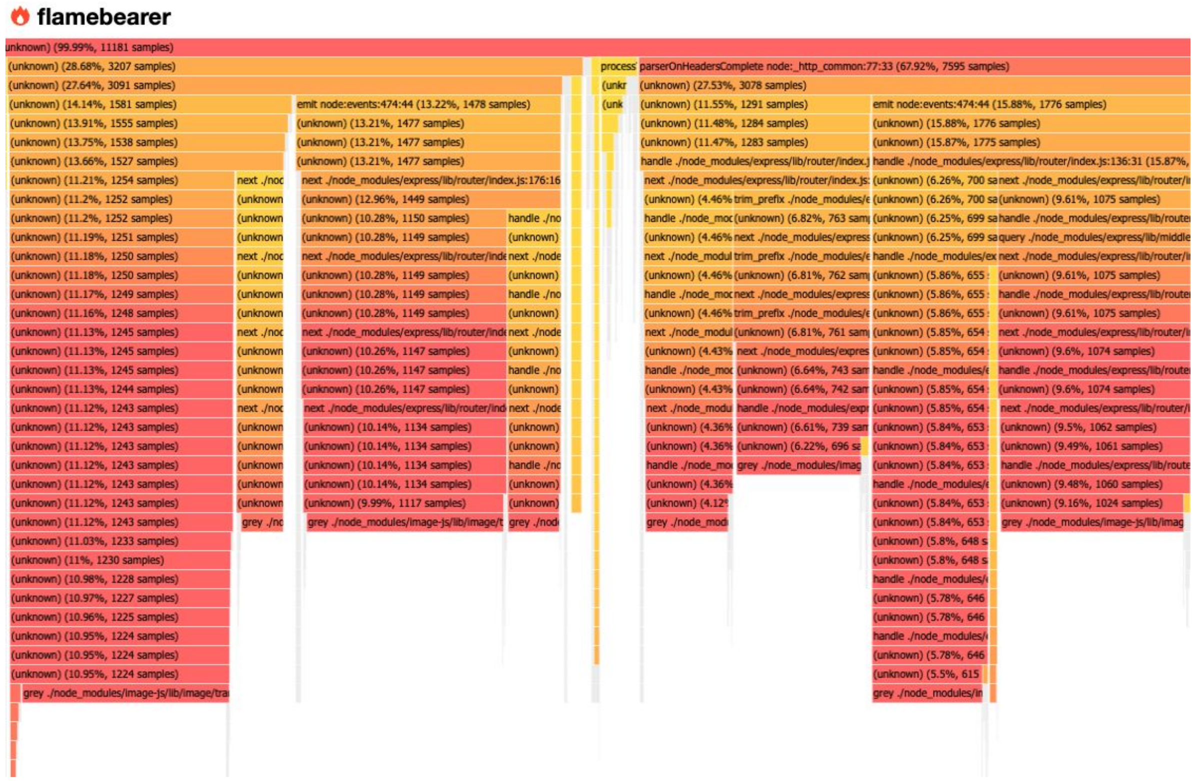


The inflection point (the point where server is no longer able to serve the requests) occurs early and stay consistent when the allowed timeout is much lower



Server CPU Monitoring / FlameGraph

An integration has been added to the Node JS server to monitor CPU performance which in conjunction with the above result can pinpoint the reason the reason for slower CPU processing



Thank You :)