CS 301
IIT Gandhinagar

Concurrent Web Servers

Project Goals

Making a Multi-Threaded Web Server

Implementing Scheduling Policies

Ensuring Security on the Server

Aditya Tripathi 18110010
Kushagra Sharma 18110091

Dishank Goel 18110052
Nishikant Parmar 18110108

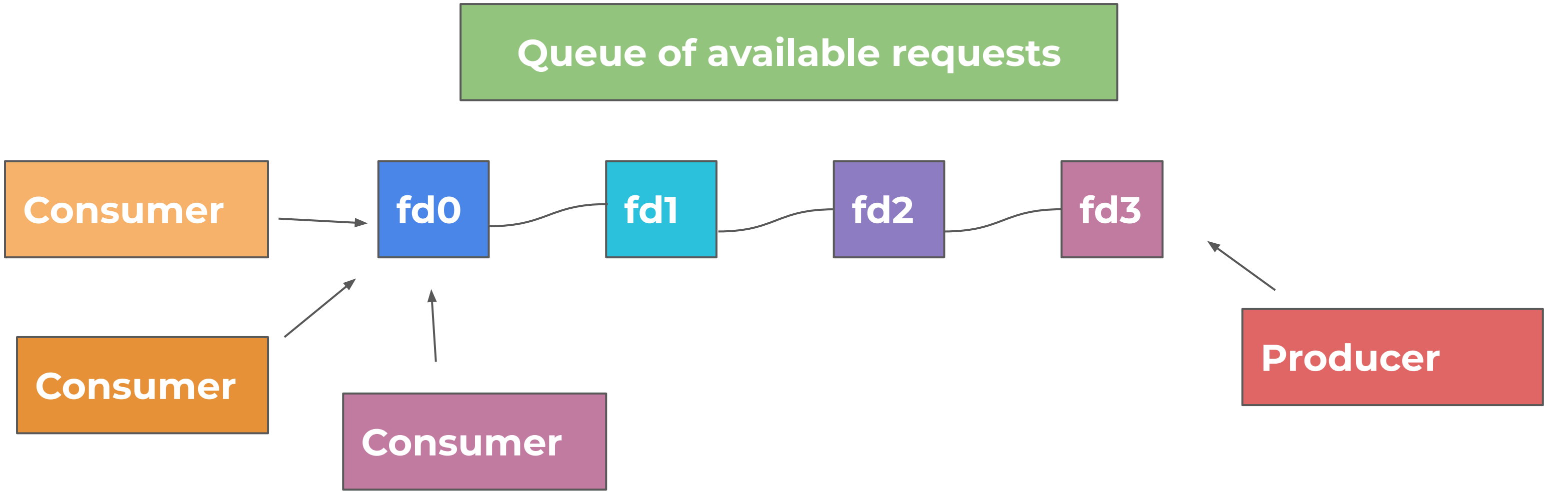
Motivation

- Server-type applications communicating with many clients demand a high degree of concurrency and performance.
- A good web server should be able to handle hundreds of thousands of concurrent connections and service tens of thousands of requests per second.
- The server should service clients in a timely, fair manner to ensure that no client starves because some other client causes the server to hang.
- Scheduling policies are required to have control over which request to serve next.
- Depending on the type of load on the server, different scheduling policies can be chosen to benefit the client-server interaction the most (for ex. Reducing the avg response time or increasing the throughput etc).

Related Work

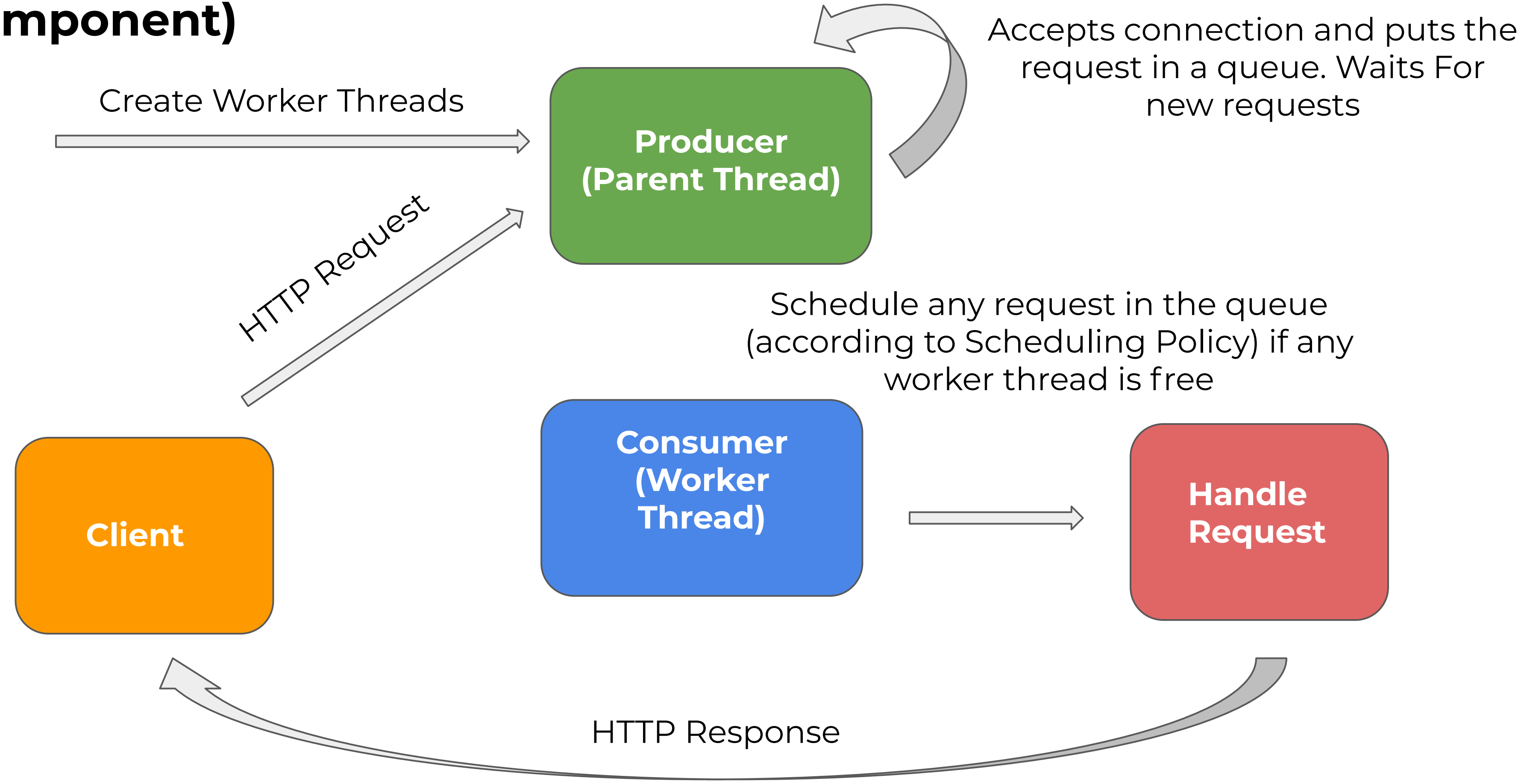
- Base Model has been taken from: OSTEP Webserver. The base webserver can only server a single client at a given time. It is a non-concurrent server and hence blocks other requests while it is serving the current one.

Concurrency: The Basic Ideology (OS Component)

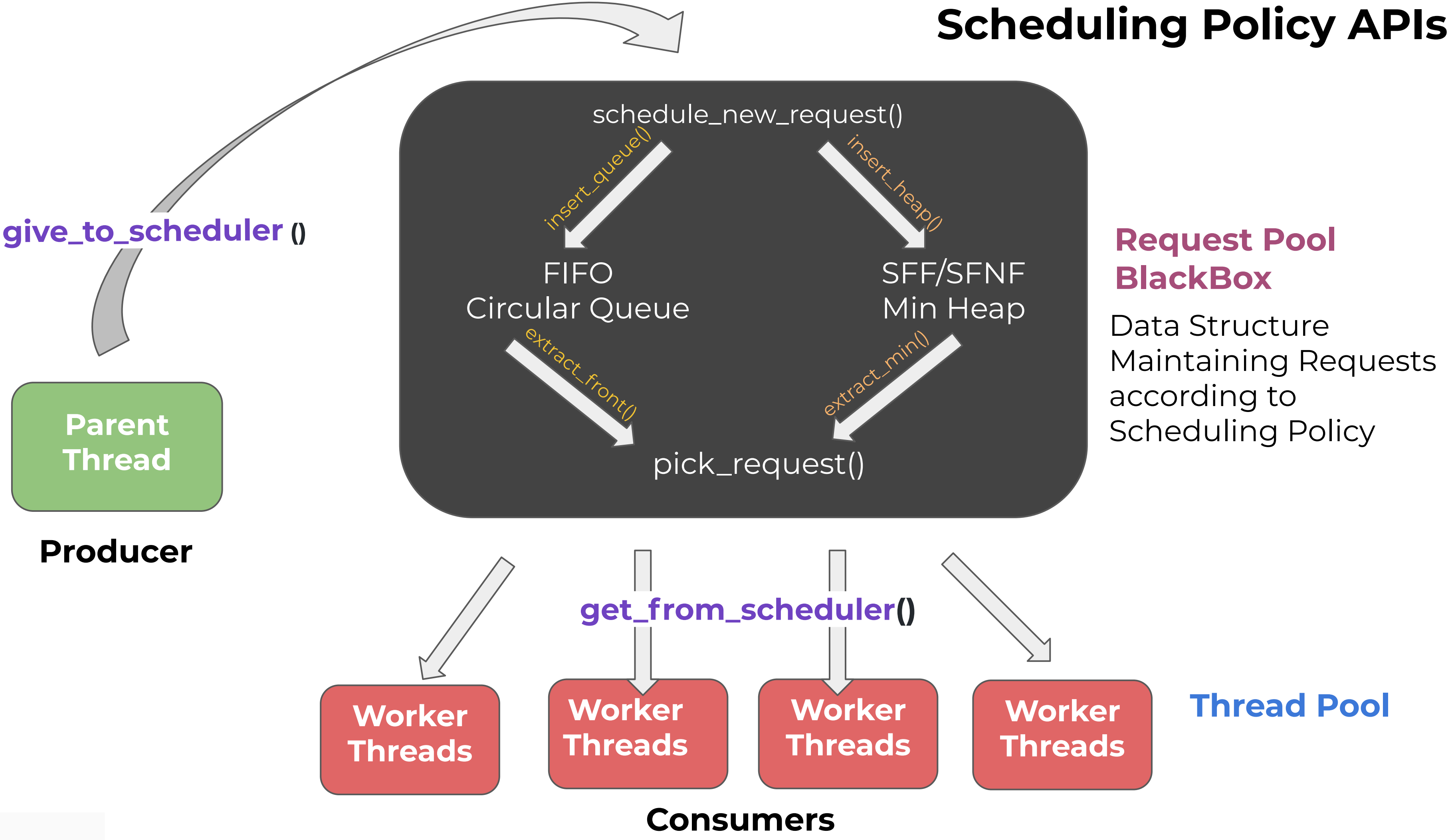


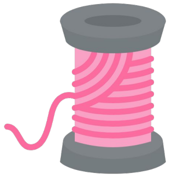
- Producer is the master server thread which adds a new request to the queue
- Each consumer is a slave server thread which handles the request available to it.

Concurrency Architecture (OS Component)



Scheduling Policy APIs





Thread Pool

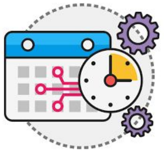
```c
typedef struct __thread_pool_t {

    int num_threads;
    pthread_t* pool;
    pthread_mutex_t LOCK;
    pthread_cond_t FILL;
    pthread_cond_t EMPTY;

} thread_pool;
```

Producer

```c
void give_to_scheduler(thread_pool* workers, scheduler* d, int conn_fd) {

    Pthread_mutex_lock(&workers->LOCK);
    while(is_scheduler_full(d)) {
        Pthread_cond_wait(&workers->FILL, &workers->LOCK);
    }
    schedule_new_request(d, conn_fd);
    Pthread_cond_signal(&workers->EMPTY);
    Pthread_mutex_unlock(&workers->LOCK);

}
```

Consumer

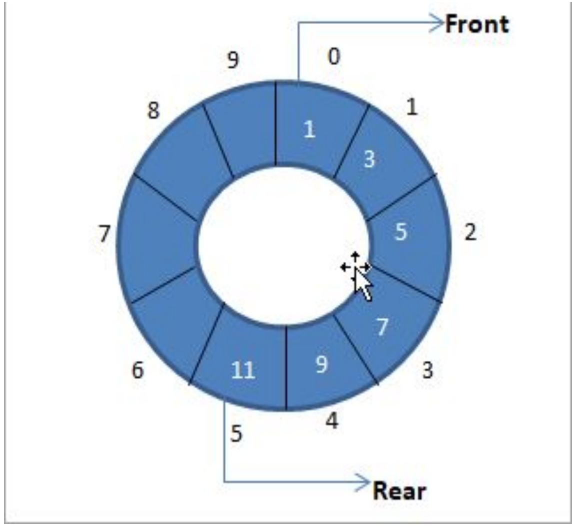
```c
int get_from_scheduler(thread_pool* workers, scheduler* d) {

    Pthread_mutex_lock(&workers->LOCK);
    while(is_scheduler_empty(d)) {
        Pthread_cond_wait(&workers->EMPTY, &workers->LOCK);
    }
    int conn_fd = pick_request(d);

    Pthread_cond_signal(&workers->FILL);
    Pthread_mutex_unlock(&workers->LOCK);

    return conn_fd;
}
```



Scheduler

```c
typedef struct __scheduler_t {

    char* policy;
    int buffer_size;
    int curr_size;
    heap* Heap;
    queue* Queue;

} scheduler;
```

Scheduling Policy - FIFO
Data Structure - Circular Queue




```c
typedef struct __queue_t {
    int max_size;
    int curr_size;
    int fill;
    int use;
    node* array;
} queue;
```

```c
typedef struct __node_t {
    int fd;
    off_t parameter;
    char* file_name;
} node;
```

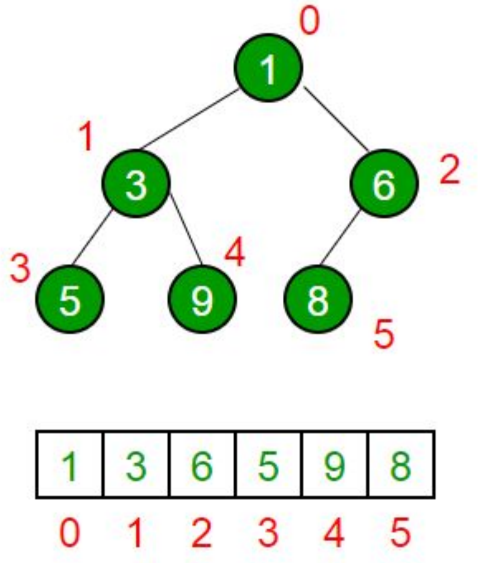
Source
softwaretestinghelp.com/cpp-circular-queue-data-structure/

Scheduling Policy - FIFO
Data Structure - Circular Queue

```c
queue* init_queue(int queue_size);
void insert_in_queue(int conn_fd, queue* Queue);
int get_from_queue(queue* Queue);
```

Scheduling Policy - SFF
Data Structure - Min Heap

```c
typedef struct __heap_t {
    int curr_size;
    int max_size;
    int by_file_name;
    node* array;
} heap;
```




```c
typedef struct __node_t {
    int fd;
    off_t parameter;
    char* file_name;
} node;
```

Source
softwaretestinghelp.com/cpp-circular-queue-data-structure/

Scheduling Policy - SFF
Data Structure - Min Heap

```
void _swap(node* x, node* y);
heap* init_heap(int heap_size, int by_file_name);
void insert_in_heap(int conn_fd, off_t parameter, char* file_name, heap* Heap);
void heapify(heap* Heap, int index);
int extract_min(heap* Heap);
int heap_comparator(heap* Heap, int i, int j);
```

Scheduling Policy - SFNF
Data Structure - Min Heap

Similar to SFF we introduced a custom comparator in heap that sorts nodes based on filename lexicographically instead of size.

Source
softwaretestinghelp.com/cpp-circular-queue-data-structure/

Security

Buffer Overflows:

- The server code heavily depended on sprintf which can be exploited to control buffer
- Combined with Format string vulnerability, could defeat ASLR and get RCE.

```
# offset for buffer overflow
payload = "/" + "A"*8028

#  Send the payload
r.sendline("GET {} HTTP/1.1".format(payload))
r.sendline(b"\r\n")
```

Local File Inclusion:

- Any file on the server could be read with directory traversal.
- Can get server's executable, getting offsets for any function.

```
# Exploiting local file inclusion
payload = '/../../../../../../../etc/passwd'
r.sendline("GET {} HTTP/1.1".format(payload))
r.sendline(b"\r\n")
```

Demo

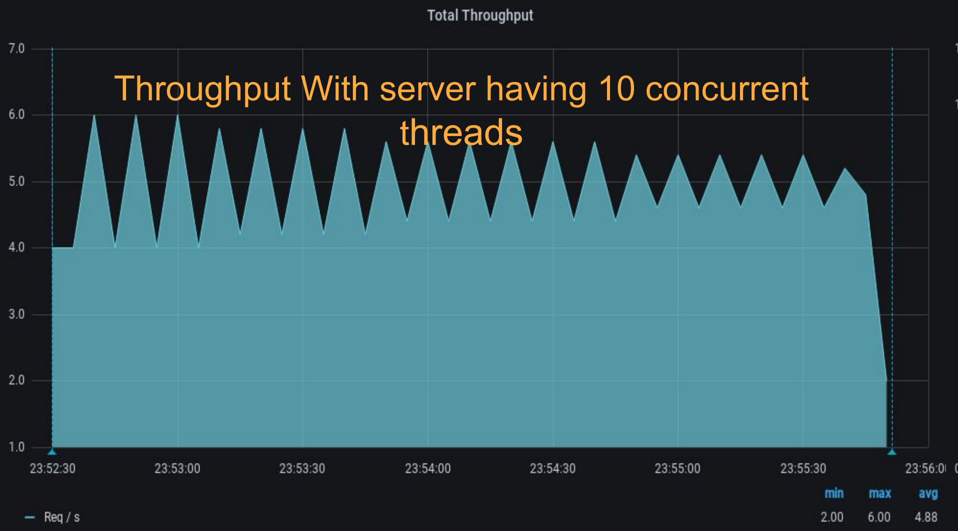
Load Testing and Analysis

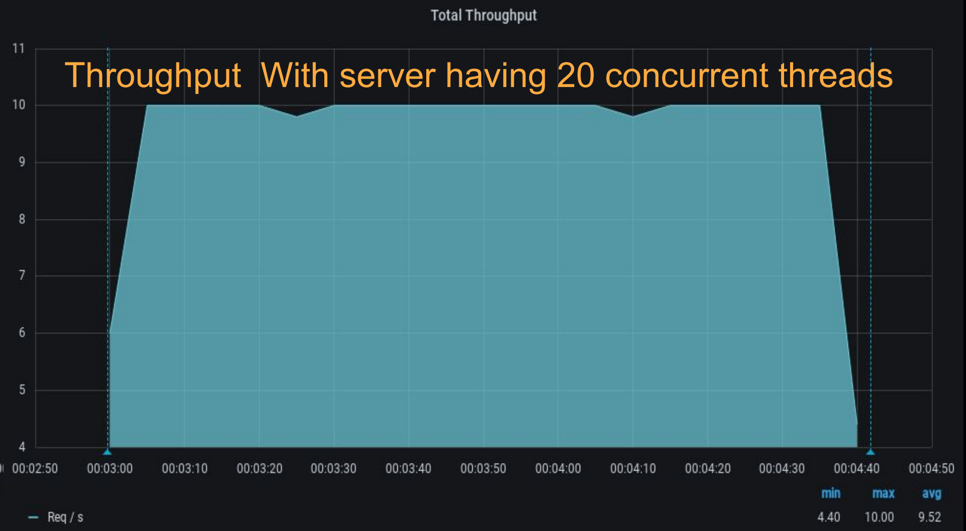
Tester Configuration:

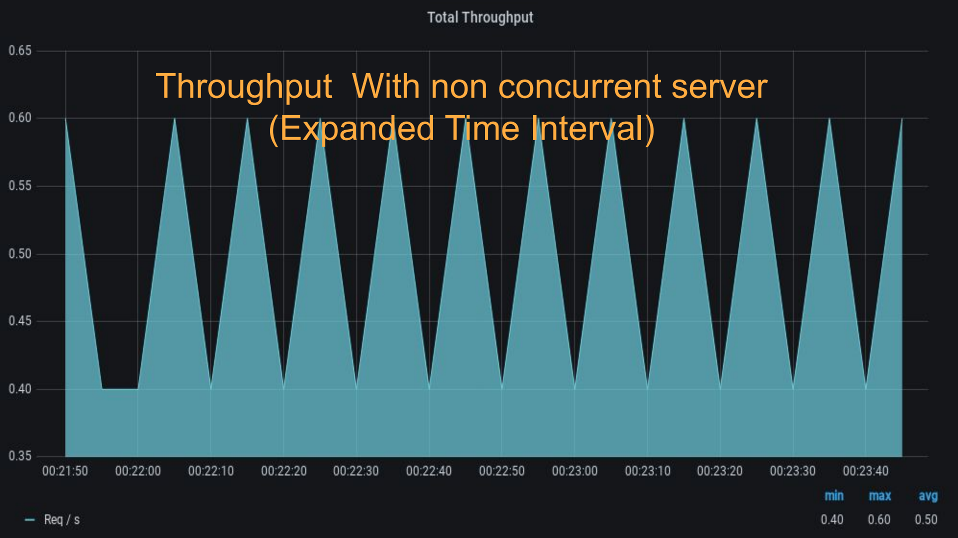
- Load test with 1000 users(threads),
- 10 concurrent users requesting the server per second with “spin.cgi”
- Program spinning for 2 seconds on server

Server Configuration:

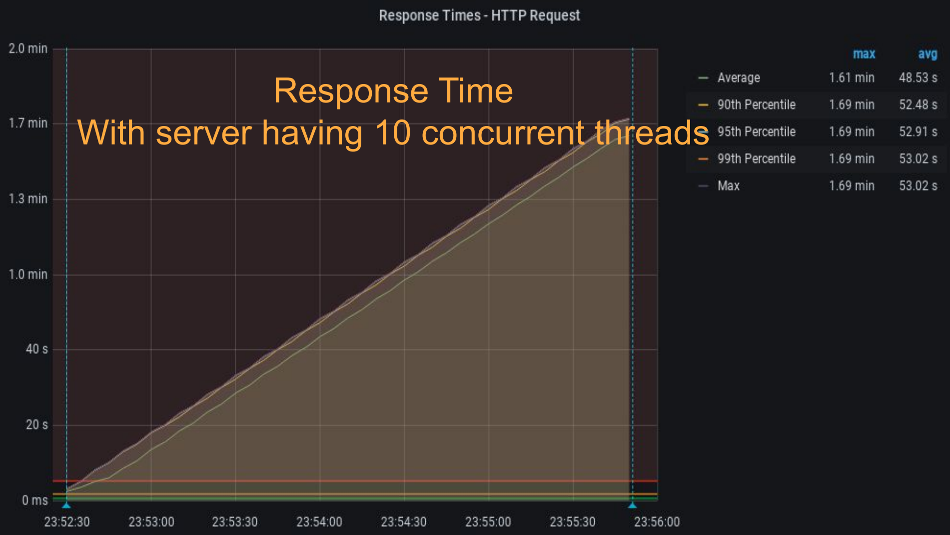
- Thread Pool: 10 worker Threads, FIFO
- Thread Pool: 20 worker Threads, FIFO
- Non Concurrent Server

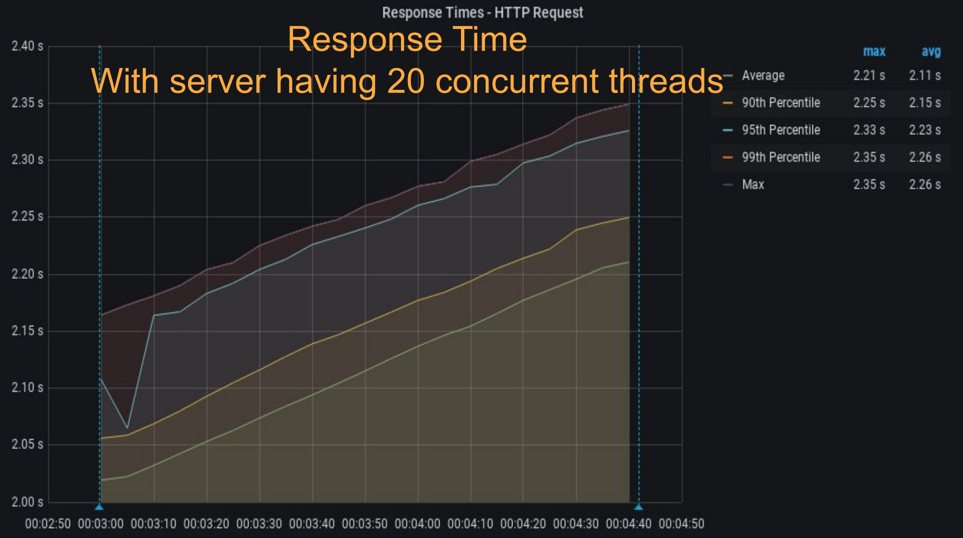


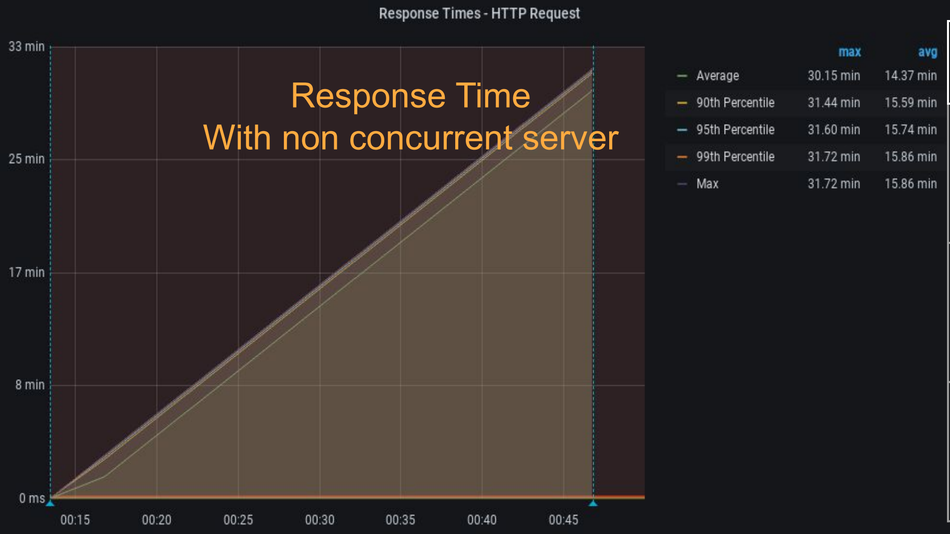




Throughput	Min(req/s)	Max(req/s)	Average(req/s)
Concurrent Server (10 threads)	2 req/s	6 req/s	4.88 req/s
Concurrent Server (20 threads)	4.40	10	9.88
Non-Concurrent server	0.4	0.6	0.5







Response Time	Time
Concurrent Server (10 threads)	48.53 sec
Concurrent Server (20 threads)	2.11 sec
Non- Concurrent server	14.37 mins

Limitations

- **HTTP versions -**
  Currently server supports only **HTTP/1.1**, but in an actual scenario different clients may be using different HTTP versions such as 0.9, 1.0, 2.0 which server is not able to handle.

- **Proxy server and cookie support -**
  Current server does set fields like **last-updated-at** on the file requested which may be used by proxy server and for setting up cookies at client.

- **Thread Scheduling by OS -**
  Scheduling policies determine which HTTP request should be handled by each of the waiting worker threads in web server. But we have no control over which thread is actually scheduled at any given time by the OS

Future Work

- **Efficient Locking Techniques -**
  Instead of locking the entire data structure that maintains requests (queue - for FIFO, heap - for SFF), we can implement hand-over-hand locking on it and get lesser overhead.

- **More Policies -**
  Implementing other policies such as Round Robin, Preemptive Priority Scheduling and comparing all the policies against various kinds of requests.

- **File Support -**
  Increasing file supports such as mp4, mp3, pdf etc.

- **Integrating with in-memory File system**

Thankyou !

Q & A

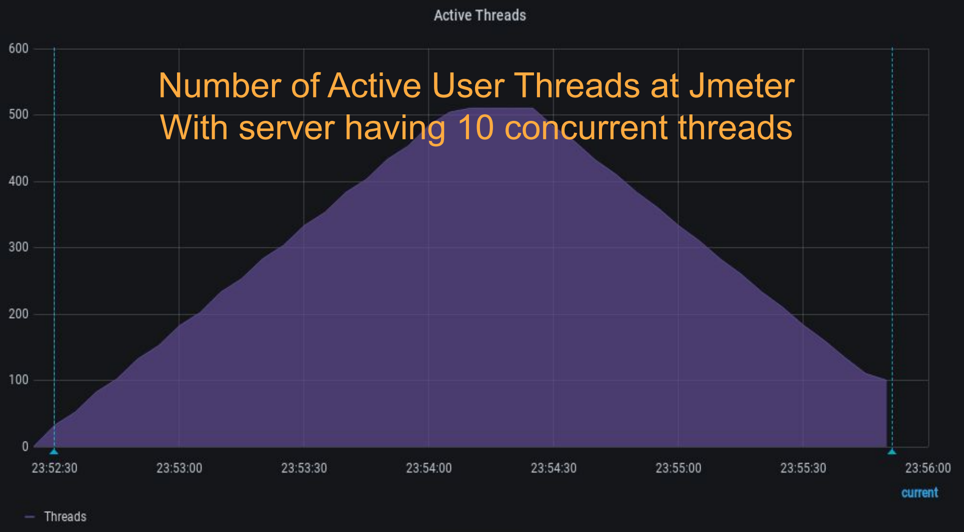
References for Images used in slides - Google Images

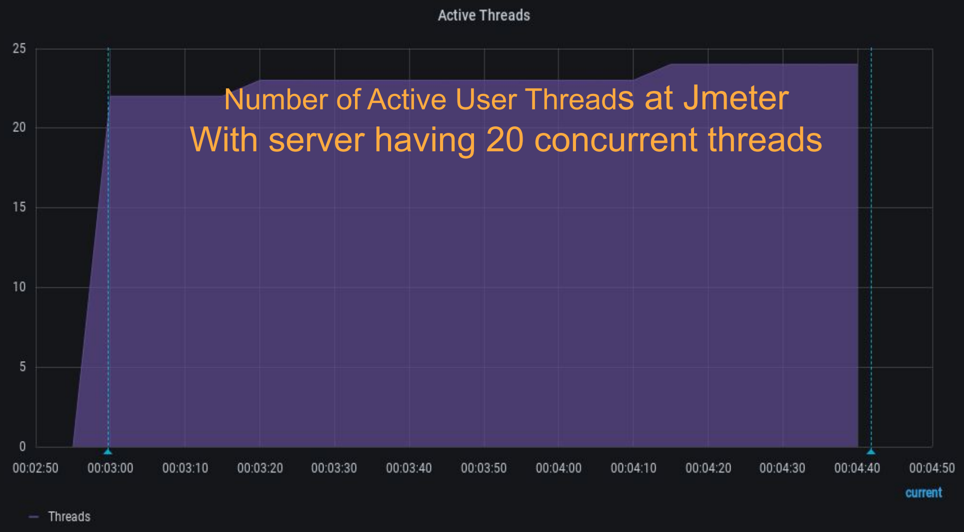
Inserting in Data Structure

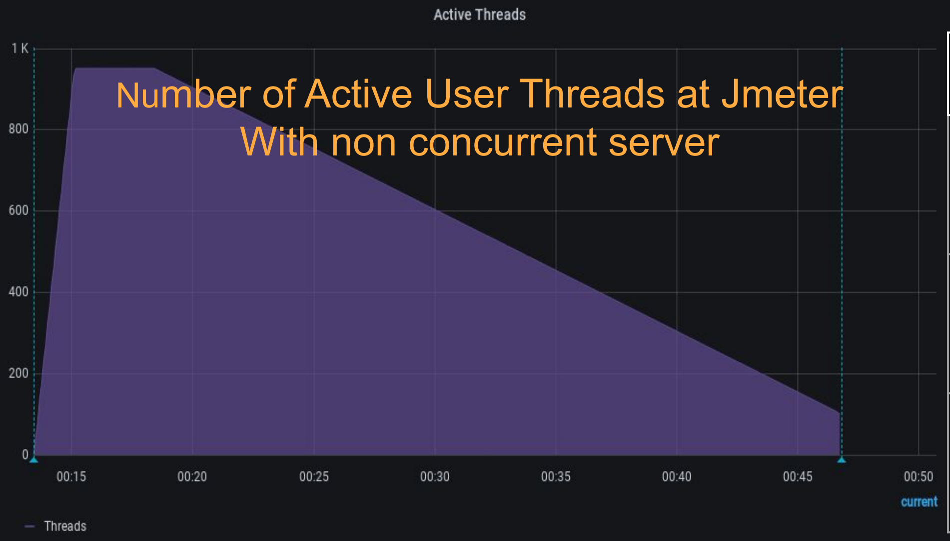
```c
void schedule_new_request(scheduler* d, int conn_fd) {

    if (strcmp(d->policy, "SFF") == 0 || strcmp("SFNF", d->policy) == 0) {

        file_prop* fileProp = request_file_properties(conn_fd);
        insert_in_heap(conn_fd, fileProp->file_size,fileProp->file_name,d->Heap);

    } else if (strcmp("FIFO", d->policy) == 0) {

        insert_in_queue(conn_fd, d->Queue);
    }
    d->curr_size++;

}
```

Extracting from Data Structure

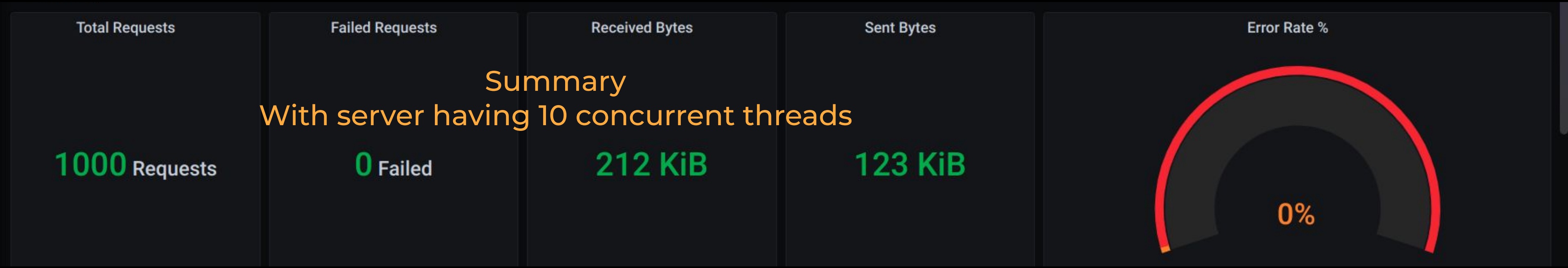
```
int pick_request(scheduler* d) {

    int conn_fd;
    if (strcmp(d->policy, "SFF") == 0 || strcmp("SFNF", d->policy) == 0) {
        conn_fd = extract_min(d->Heap);
    } else if (strcmp("FIFO", d->policy) == 0) {
        conn_fd = get_from_queue(d->Queue);
    }
    d->curr_size--;
    return conn_fd;
}
```

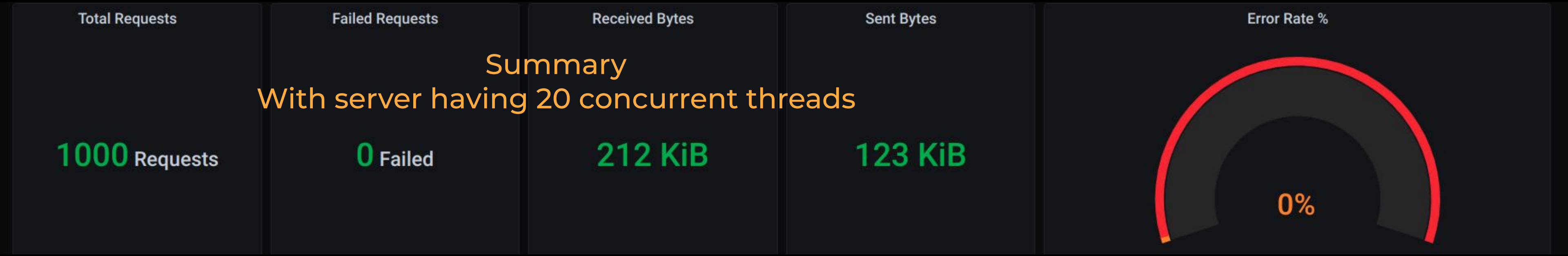


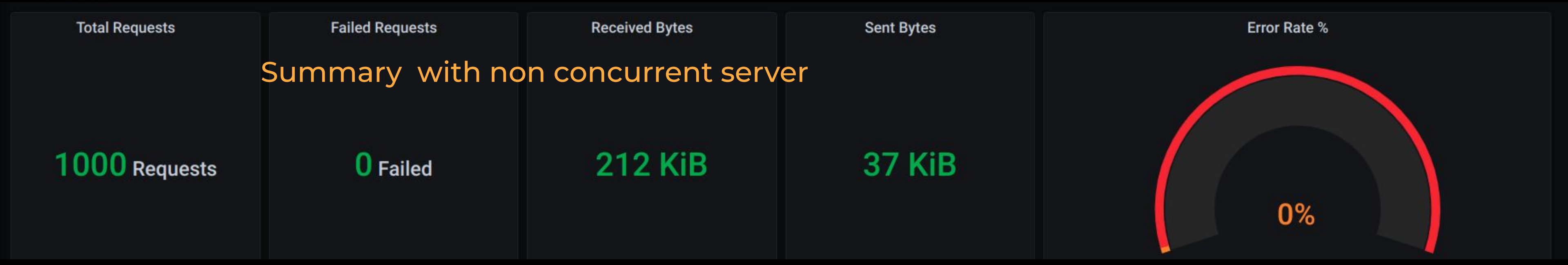




Active Threads	Max Active Threads
Concurrent Server (10 threads)	500
Concurrent Server (20 threads)	20
Non- Concurrent server	1000


Total Requests
Failed Requests
Received Bytes
Sent Bytes
Error Rate %
Summary
With server having 10 concurrent threads
1000 Requests
0 Failed
212 KiB
123 KiB
0%

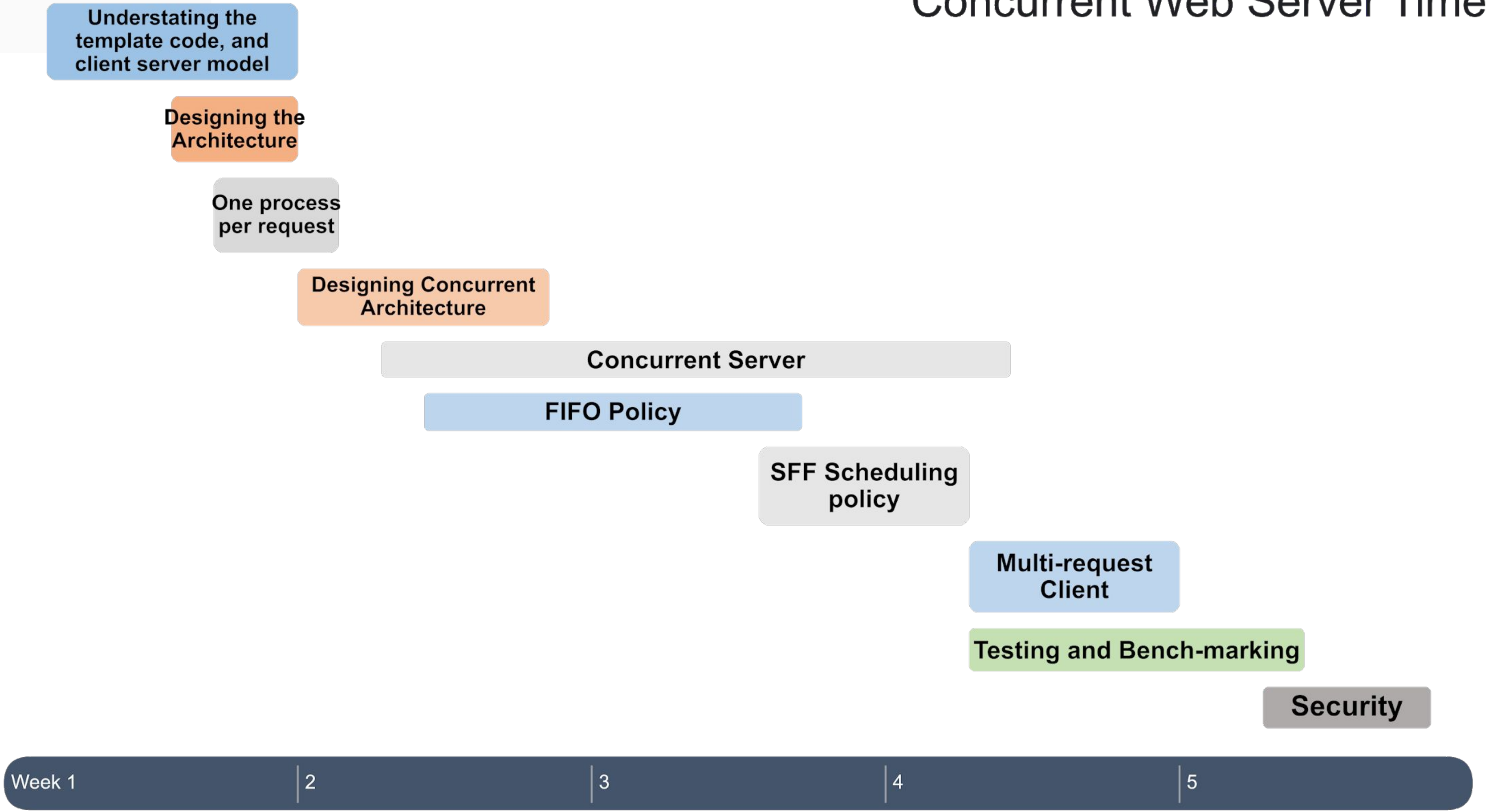

Total Requests
Failed Requests
Received Bytes
Sent Bytes
Error Rate %
Summary
With server having 20 concurrent threads
1000 Requests
0 Failed
212 KiB
123 KiB
0%


Total Requests
Failed Requests
Received Bytes
Sent Bytes
Error Rate %
Summary with non concurrent server
1000 Requests
0 Failed
212 KiB
37 KiB
0%

```c
int listen_fd = open_listen_fd_or_die(port);
    while (1) {
        struct sockaddr_in client_addr;
        int client_len = sizeof(client_addr);
        int conn_fd = accept_or_die(listen_fd, (sockaddr_t *) &client_addr, (socklen_t *) &client_len);
        int rc = fork();
        if(rc < 0)
        {
            perror("Cannot fork\n");
            exit(0);
        }
        if(rc == 0)
        {
            request_handle(conn_fd);
             close_or_die(conn_fd);
            exit(1);
        }
        close_or_die(conn_fd);
    }
    return 0;
```

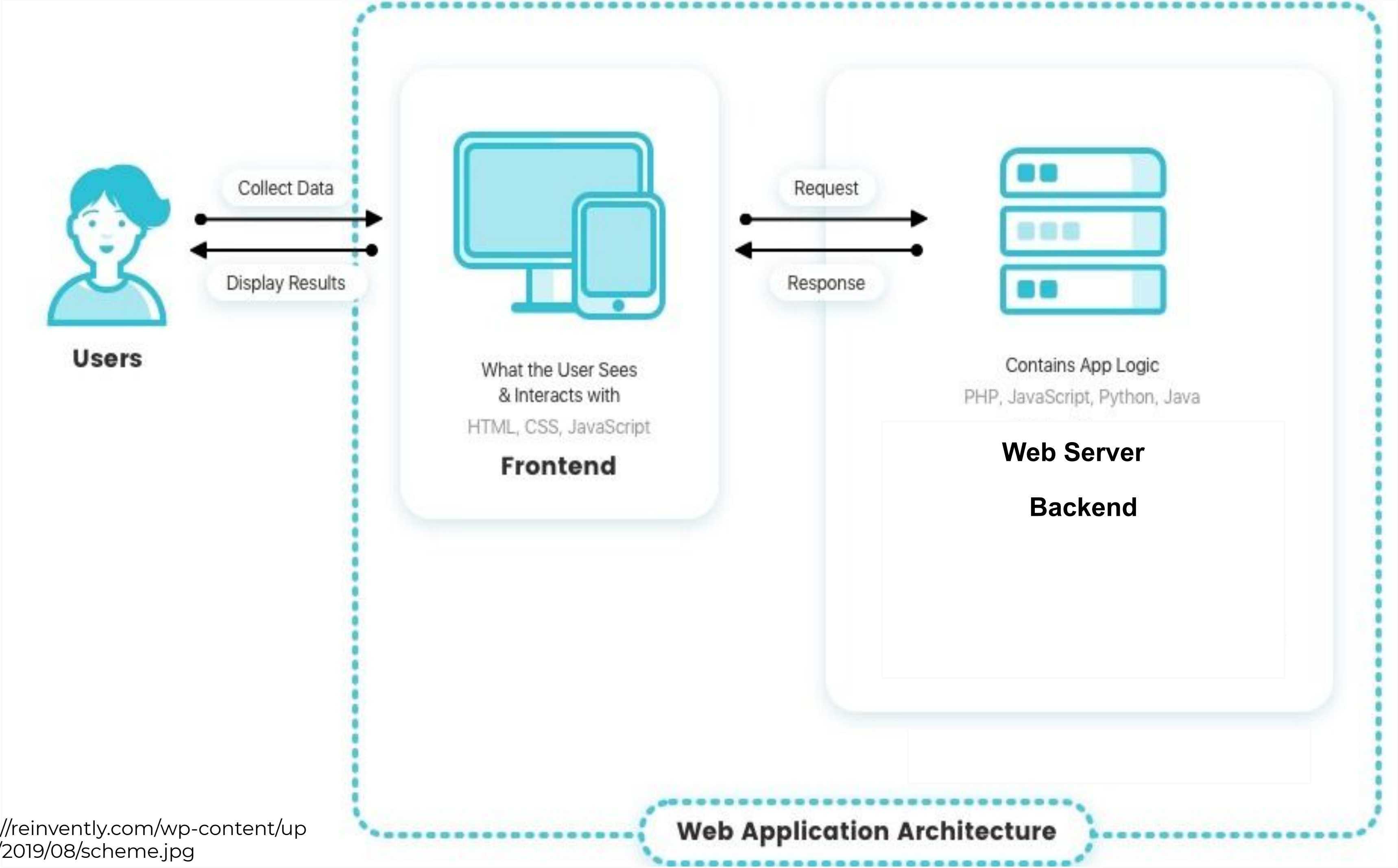
CODE!

Concurrent Web Server Timeline



Contents

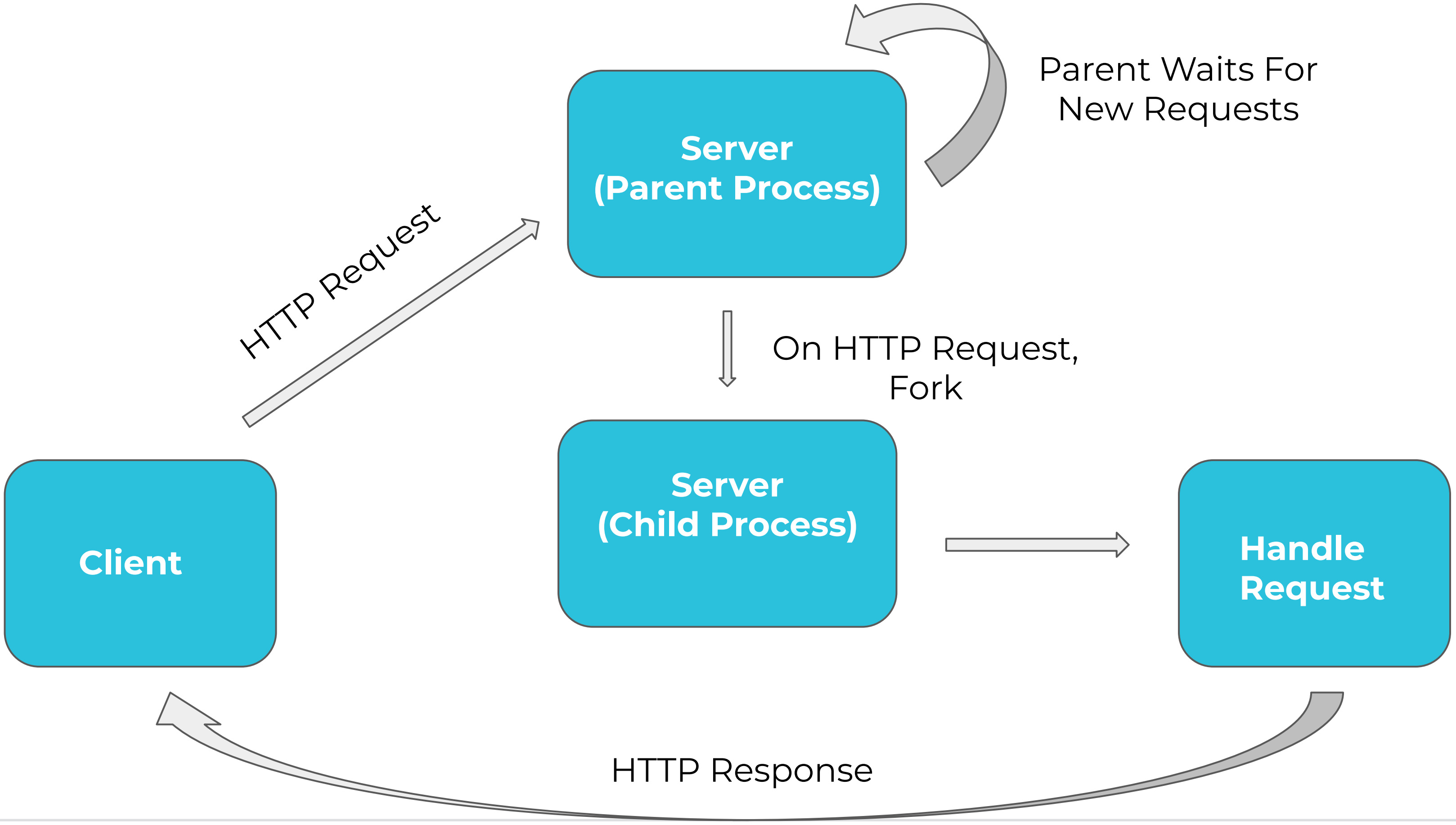
- One Process Per Request
- Concurrency
- Scheduling Policies and Scheduler
- First In First Out (FIFO)
- Shortest File First (SFF)
- Multi-Request Client
- Benchmarking
- Load Testing and Analysis



https://reinvently.com/wp-content/uploads/2019/08/scheme.jpg

One Process Per Request

One Process Per Tab Architecture



Multi-Request Client

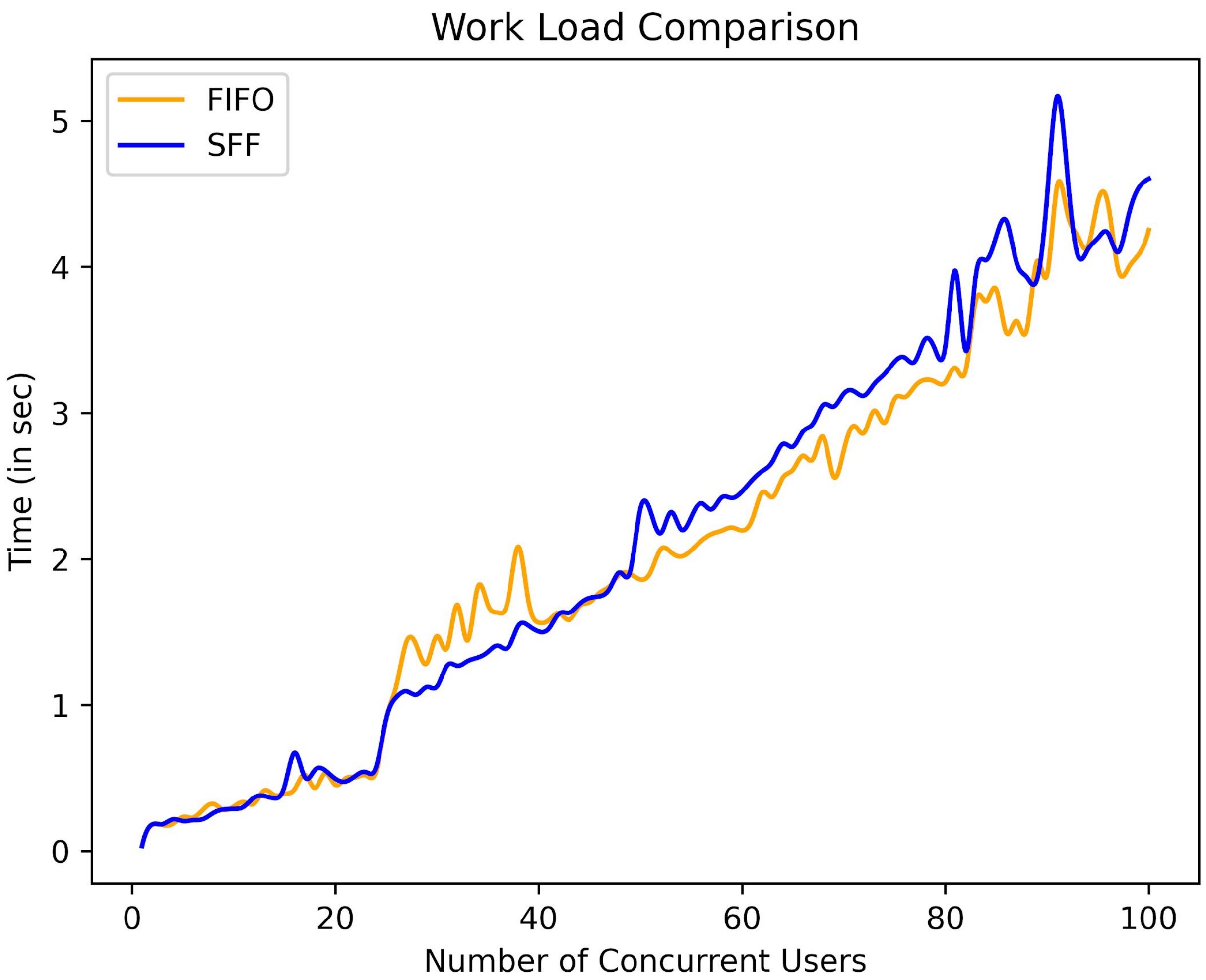
```
for(int i = 0; i < concur_clients; i++) {

    client_data *d = malloc(sizeof(client_data));
    if(d == NULL) {
        printf("Could not create request for: %s\n", argv[3 + i]);
        continue;
    }
    d->host = host;
    d->port = port;
    d->filename = argv[4 + i];
    pthread_create(&threads[i], NULL, single_client, (void *)d);

}
for(int i = 0; i < concur_clients; i++) {
    pthread_join(threads[i], NULL);
}
```

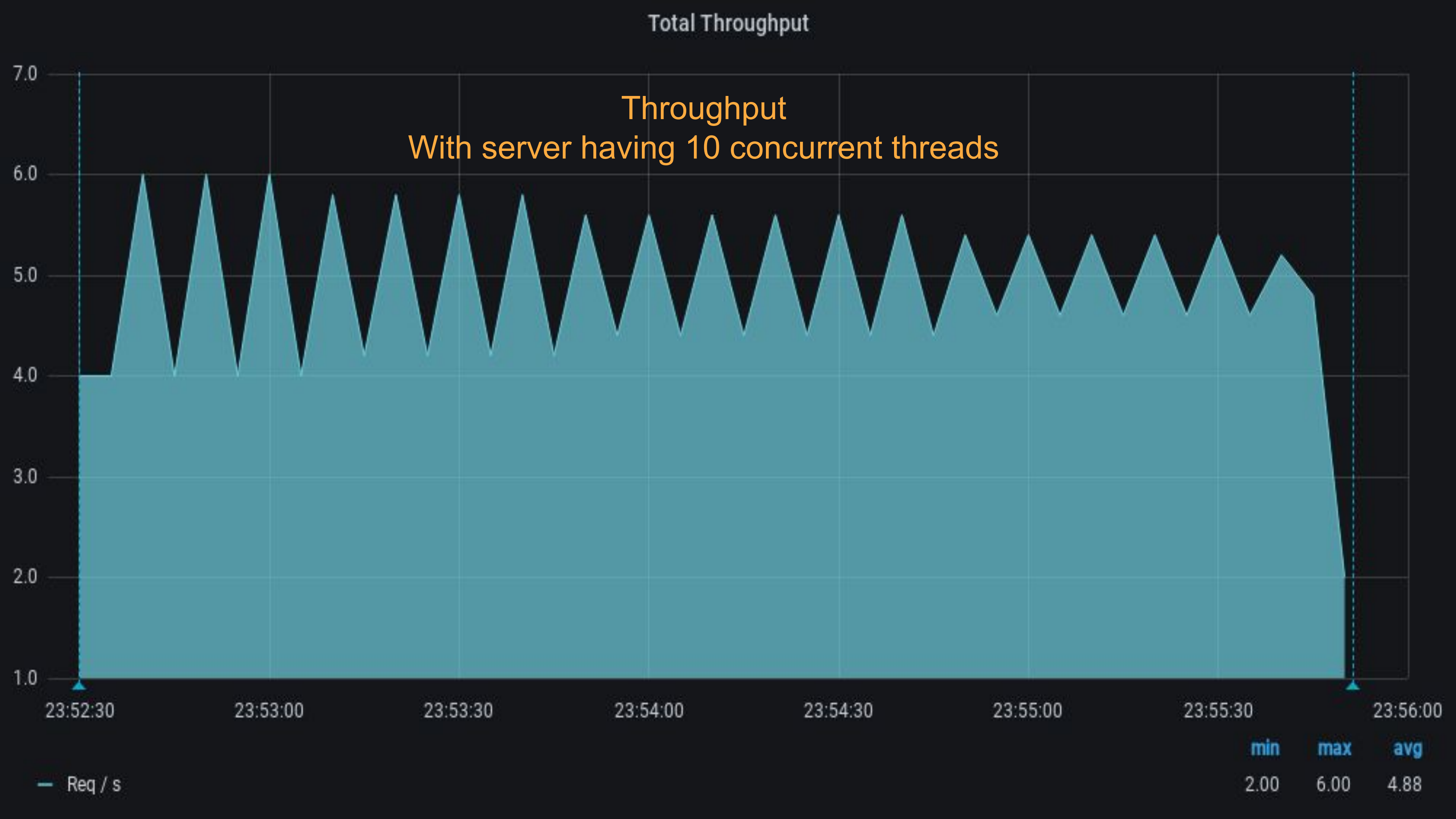
Benchmarking:
SFF vs FIFO Comparison

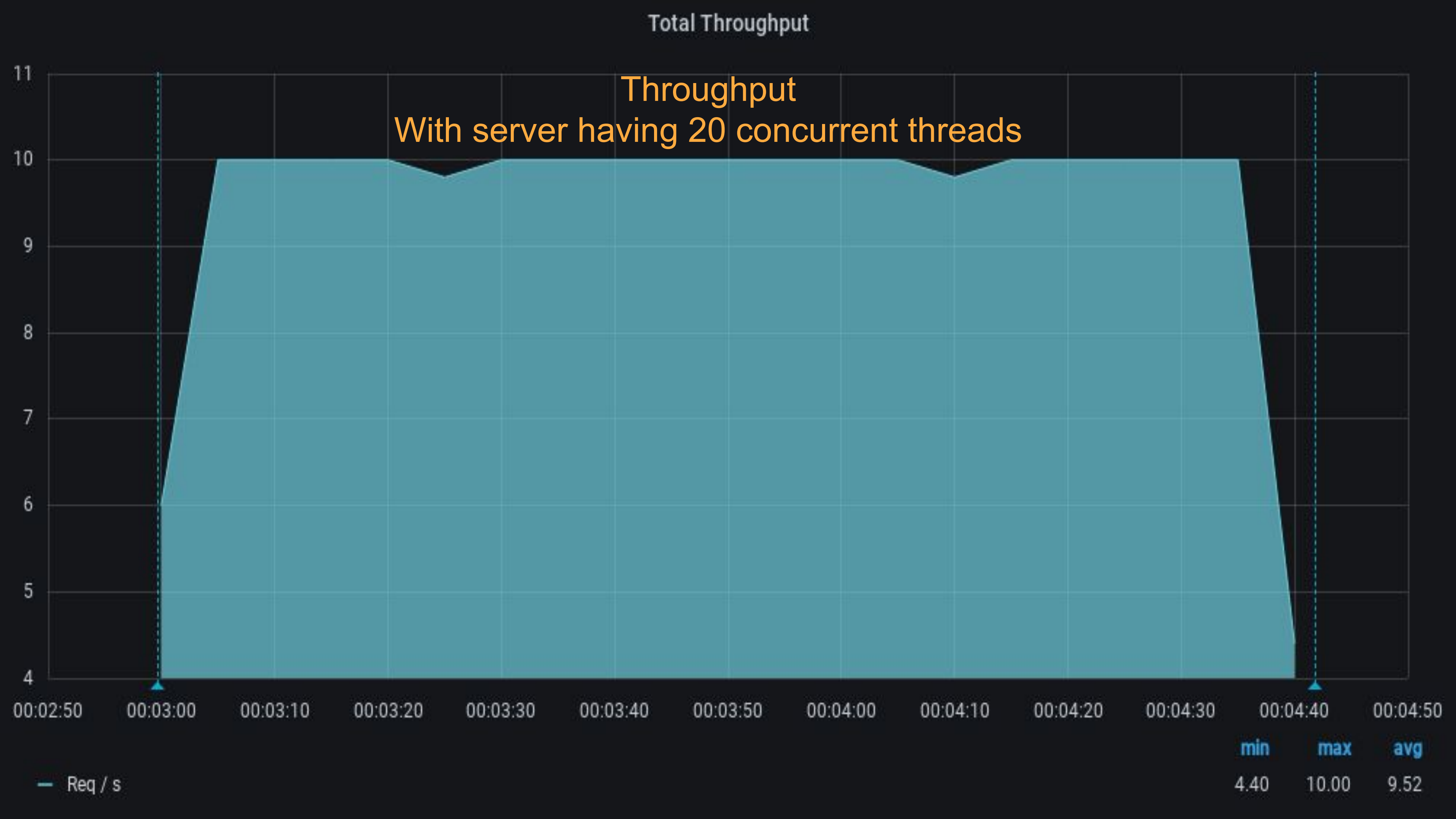
Configuration:

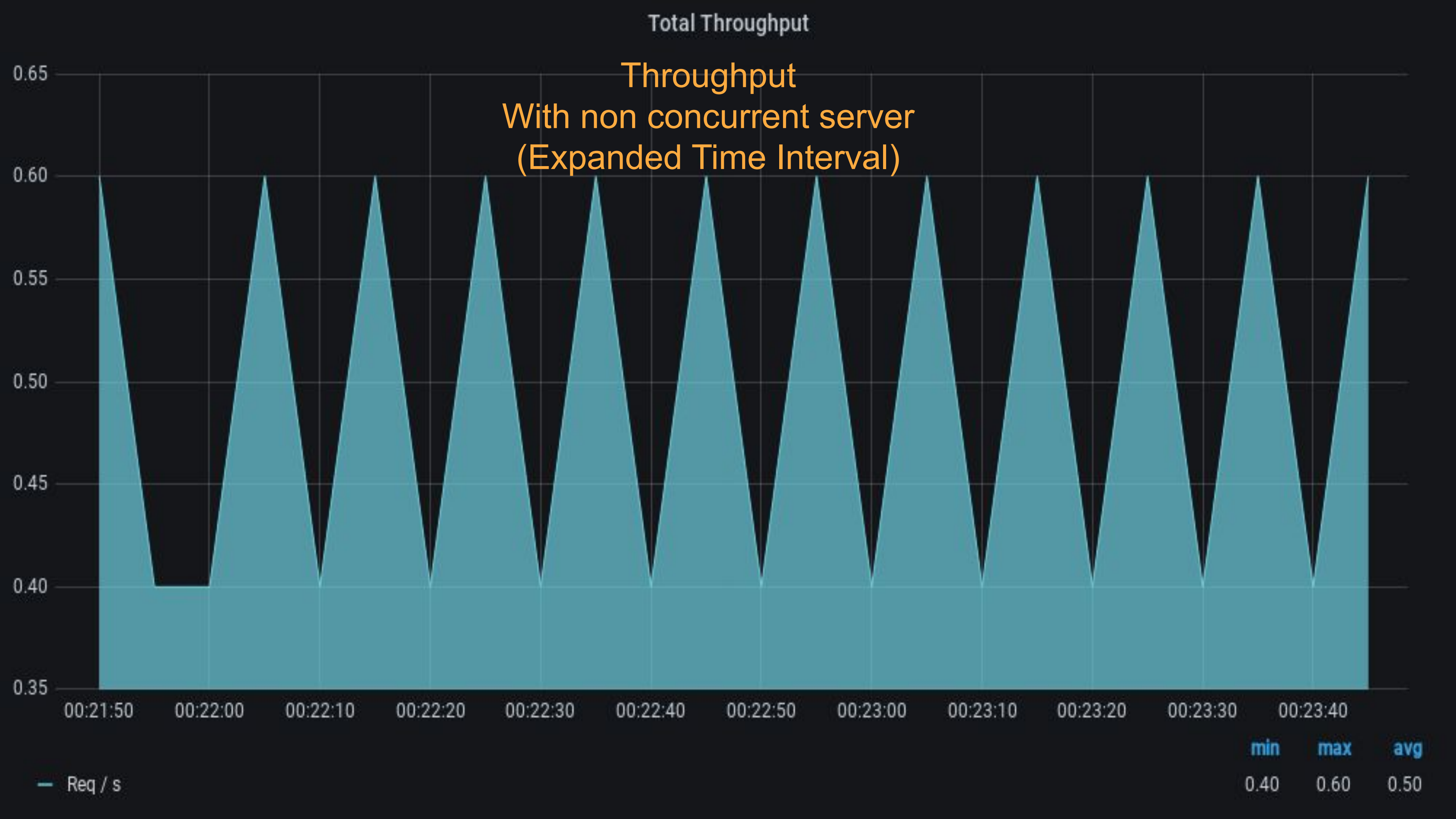
- 100 HTML loads with random sizes
- Thread Pool 50 worker Threads
- Automated Script to request the server through multi-request client
- Requests are sent in linearly increasing fashion

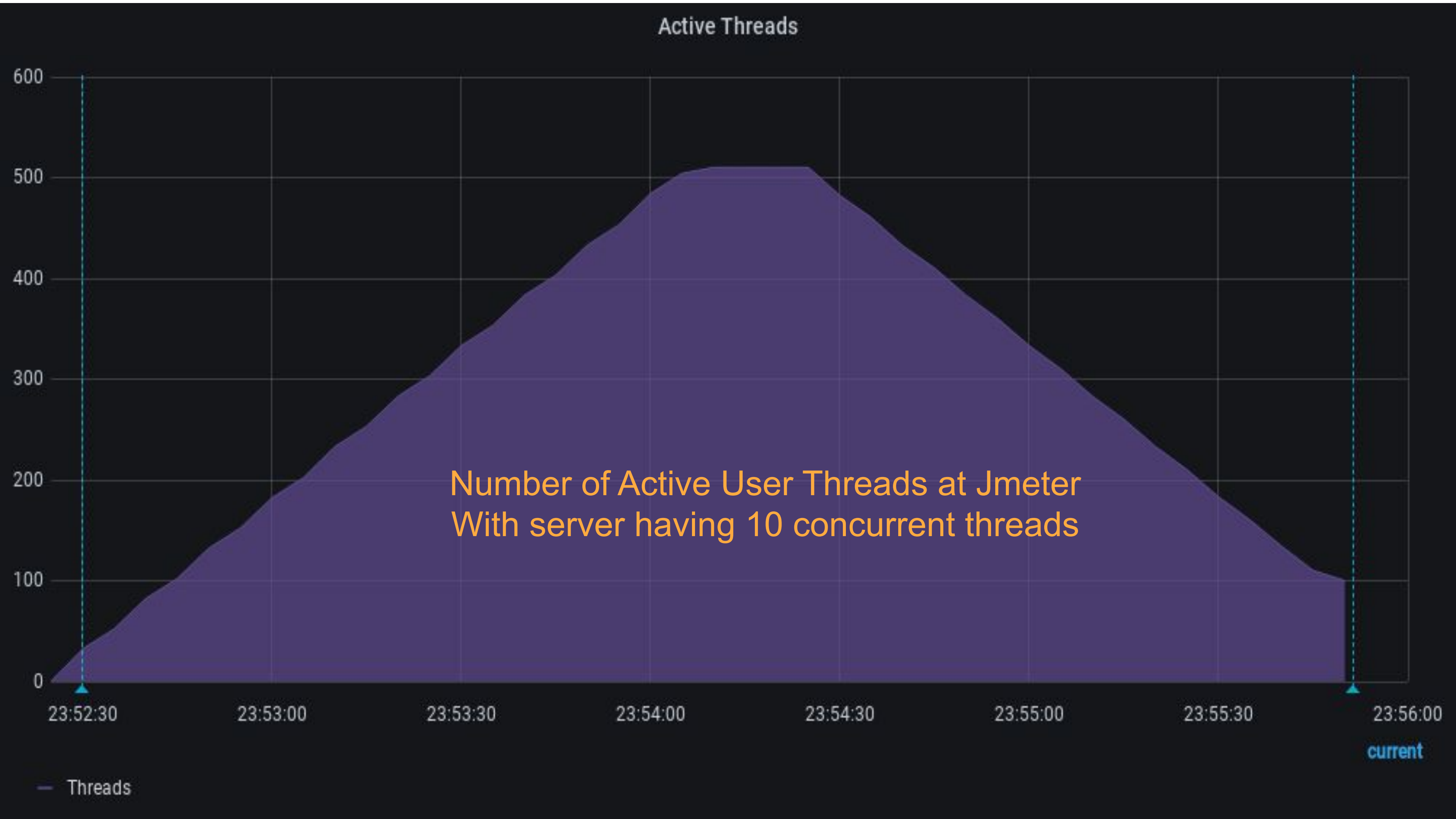
```python
NUMBER_OF_CONCURRENT_USERS = 100

command = "./wclient localhost 5000 1"
for i in range(NUMBER_OF_CONCURRENT_USERS):
    command += " /benchmarking/load/"+str(i+1) + ".html "
    print(command)
    process = subprocess.Popen(command, shell=True)
    process.wait()
```


Work Load Comparison
FIFO
SFF
Time (in sec)
Number of Concurrent Users
0
1
2
3
4
5
0
20
40
60
80
100

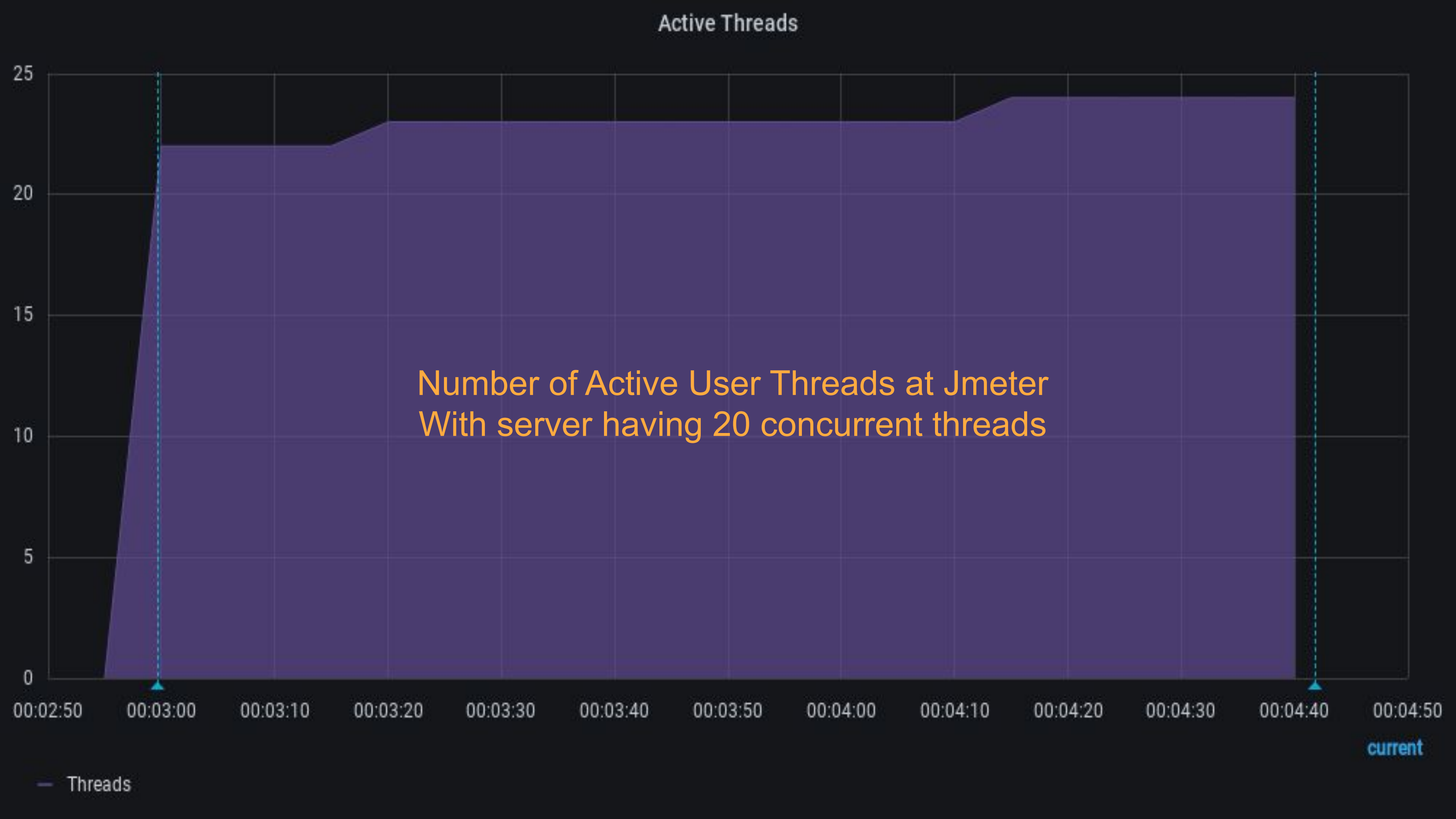

Total Throughput
Throughput
With server having 10 concurrent threads
7.0
6.0
5.0
4.0
3.0
2.0
1.0
23:52:30
23:53:00
23:53:30
23:54:00
23:54:30
23:55:00
23:55:30
23:56:00
min
max
avg
Req / s
2.00
6.00
4.88


Total Throughput
Throughput
With server having 20 concurrent threads
11
10
9
8
7
6
5
4
00:02:50
00:03:00
00:03:10
00:03:20
00:03:30
00:03:40
00:03:50
00:04:00
00:04:10
00:04:20
00:04:30
00:04:40
00:04:50
min
max
avg
Req / s
4.40
10.00
9.52

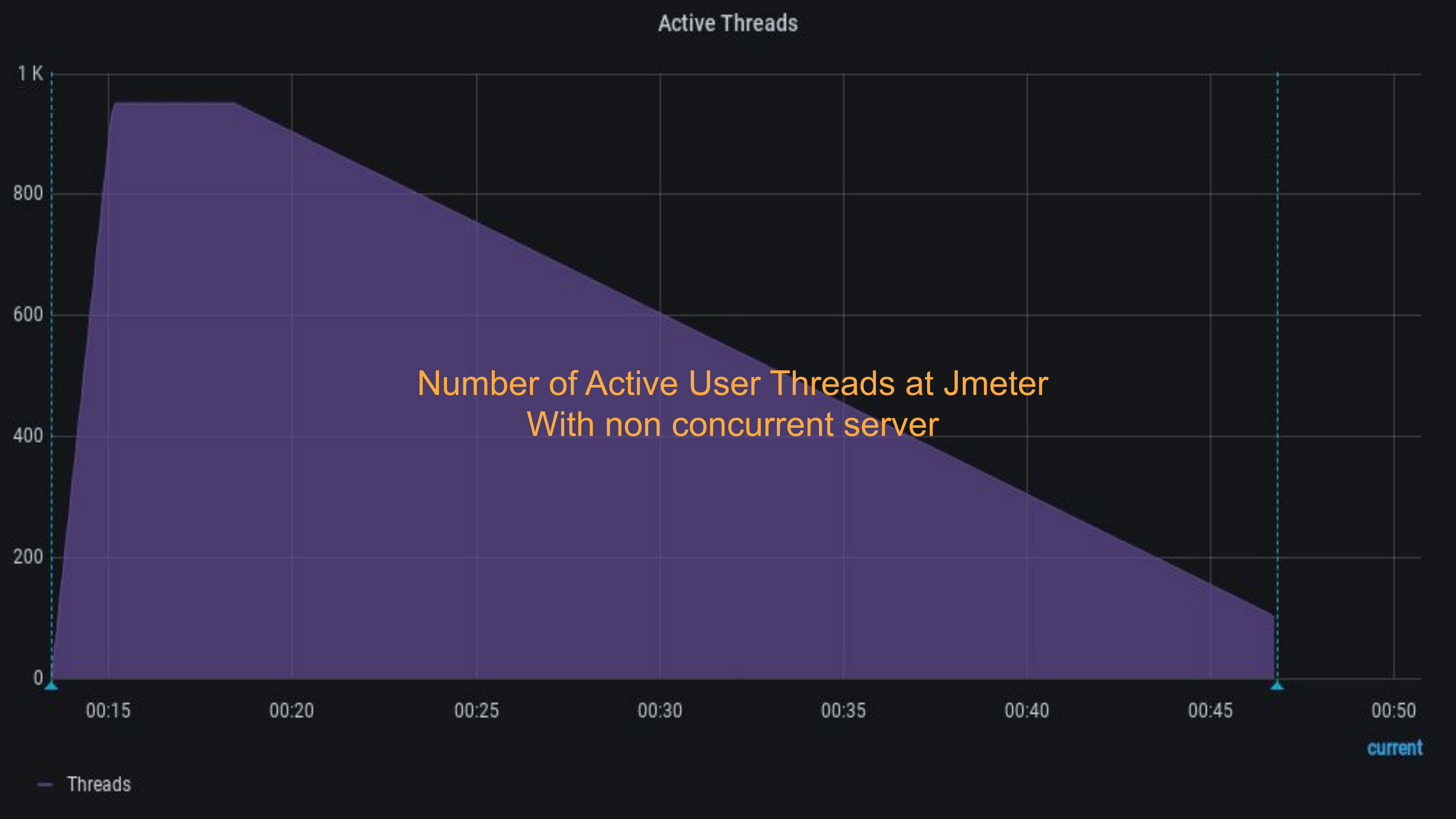

Total Throughput
Throughput
With non concurrent server
(Expanded Time Interval)
0.65
0.60
0.55
0.50
0.45
0.40
0.35
00:21:50
00:22:00
00:22:10
00:22:20
00:22:30
00:22:40
00:22:50
00:23:00
00:23:10
00:23:20
00:23:30
00:23:40
min
max
avg
Req / s
0.40
0.60
0.50



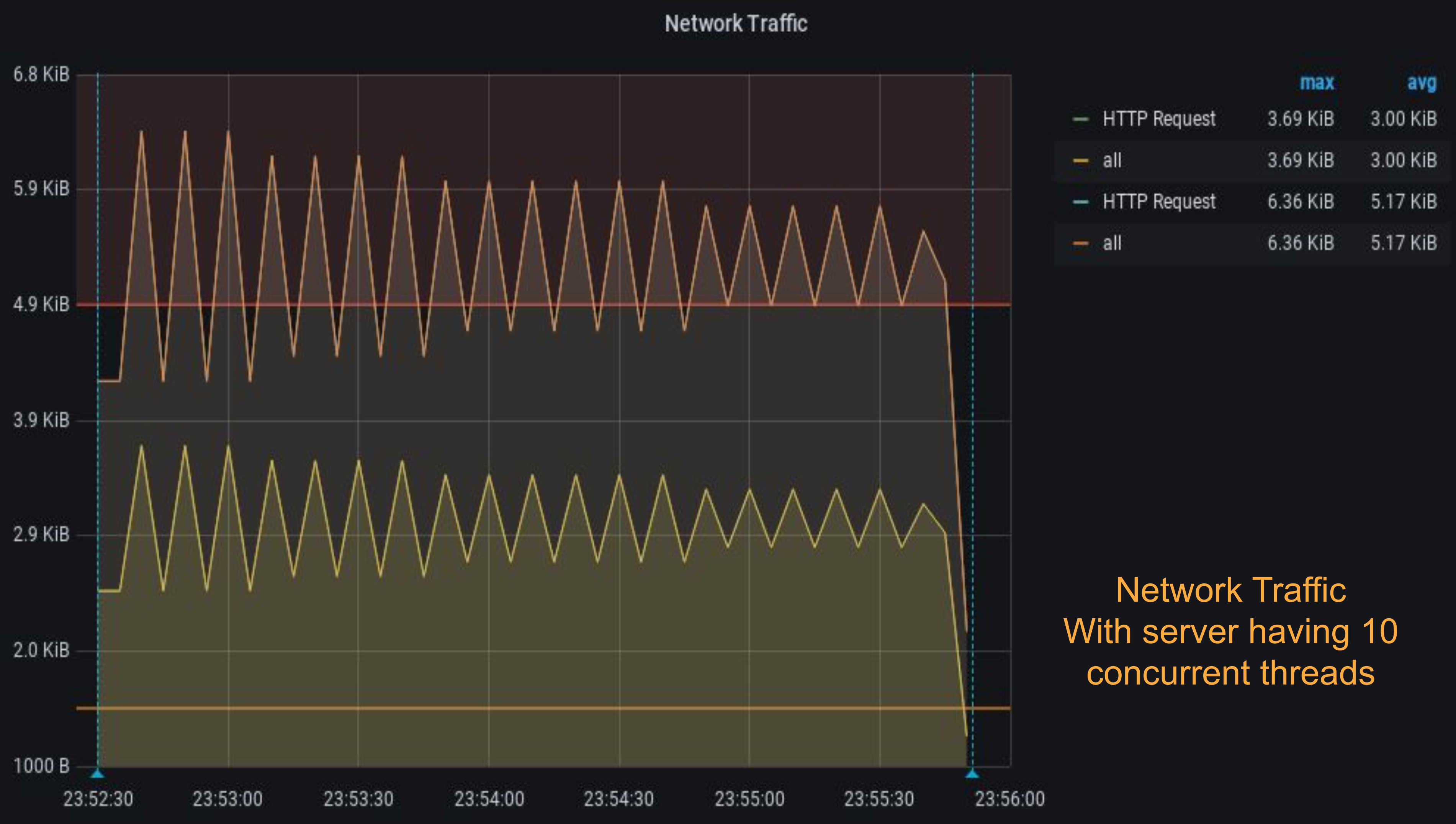
Active Threads
600
500
400
300
200
100
0
23:52:30
23:53:00
23:53:30
23:54:00
23:54:30
23:55:00
23:55:30
23:56:00
current
Threads
Number of Active User Threads at Jmeter
With server having 10 concurrent threads



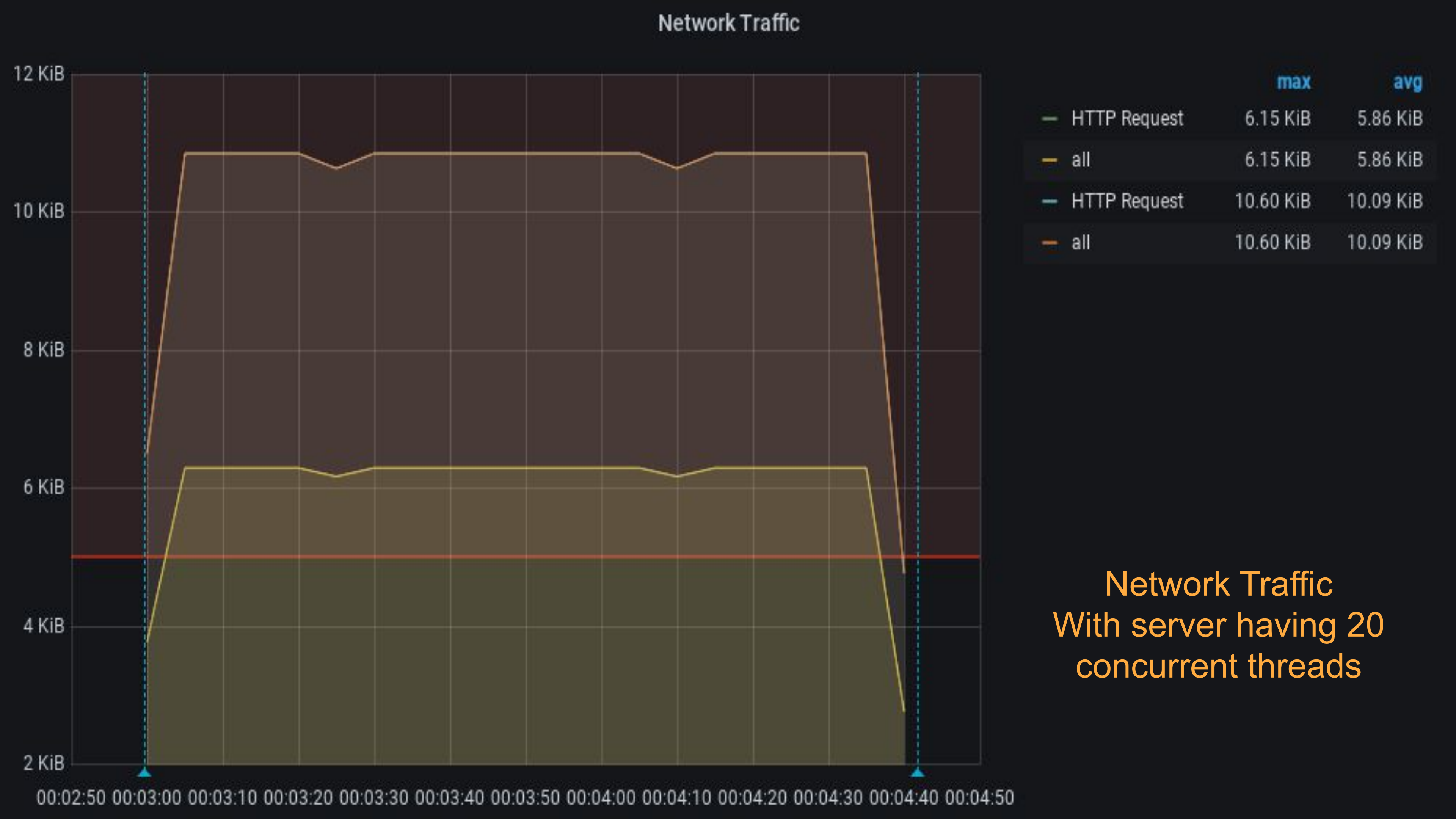
Active Threads
25
20
15
10
5
0
00:02:50
00:03:00
00:03:10
00:03:20
00:03:30
00:03:40
00:03:50
00:04:00
00:04:10
00:04:20
00:04:30
00:04:40
00:04:50
current
Threads
Number of Active User Threads at Jmeter
With server having 20 concurrent threads



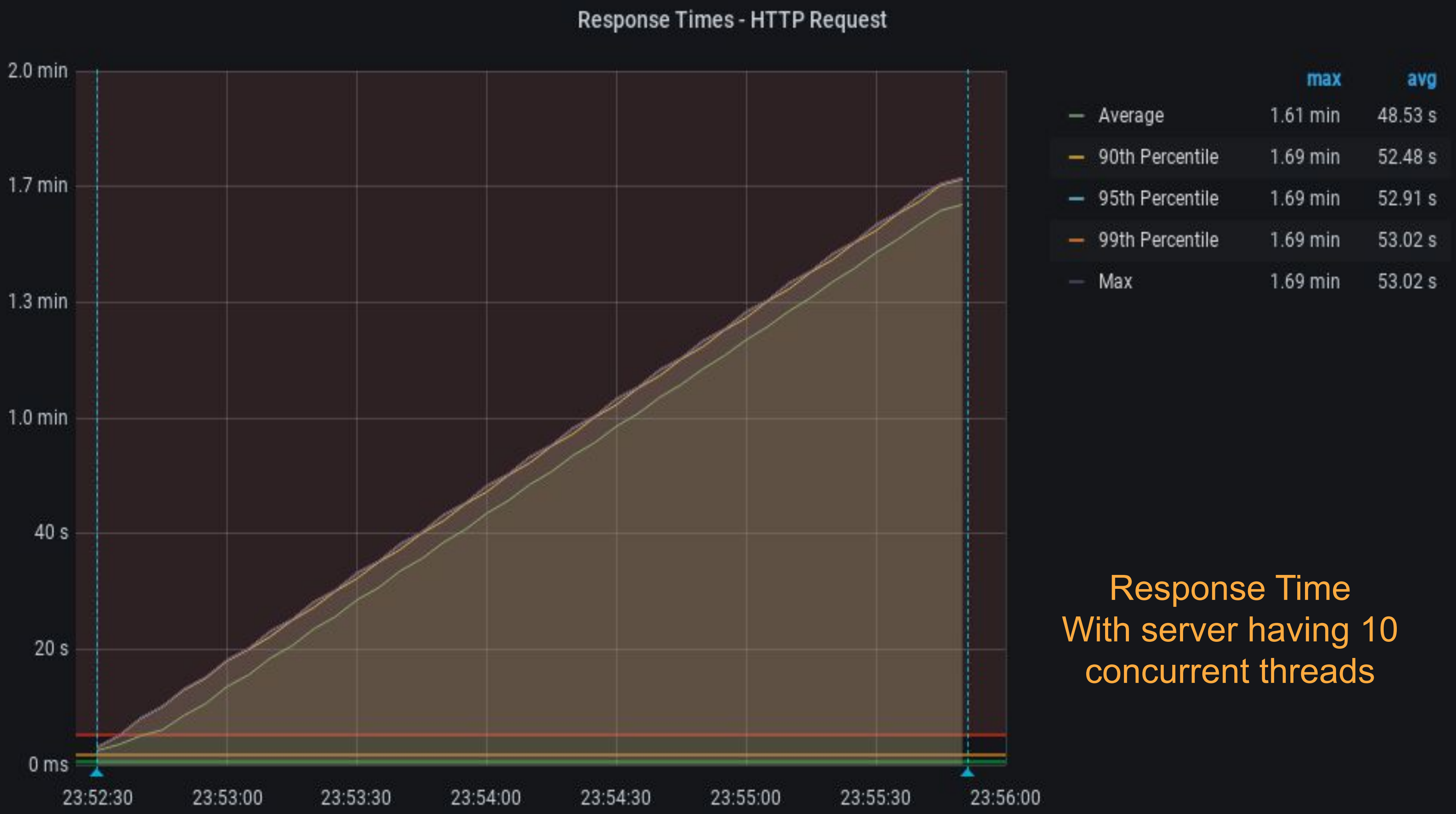
Active Threads
1 K
800
600
400
200
0
Number of Active User Threads at Jmeter
With non concurrent server
00:15
00:20
00:25
00:30
00:35
00:40
00:45
00:50
current
Threads



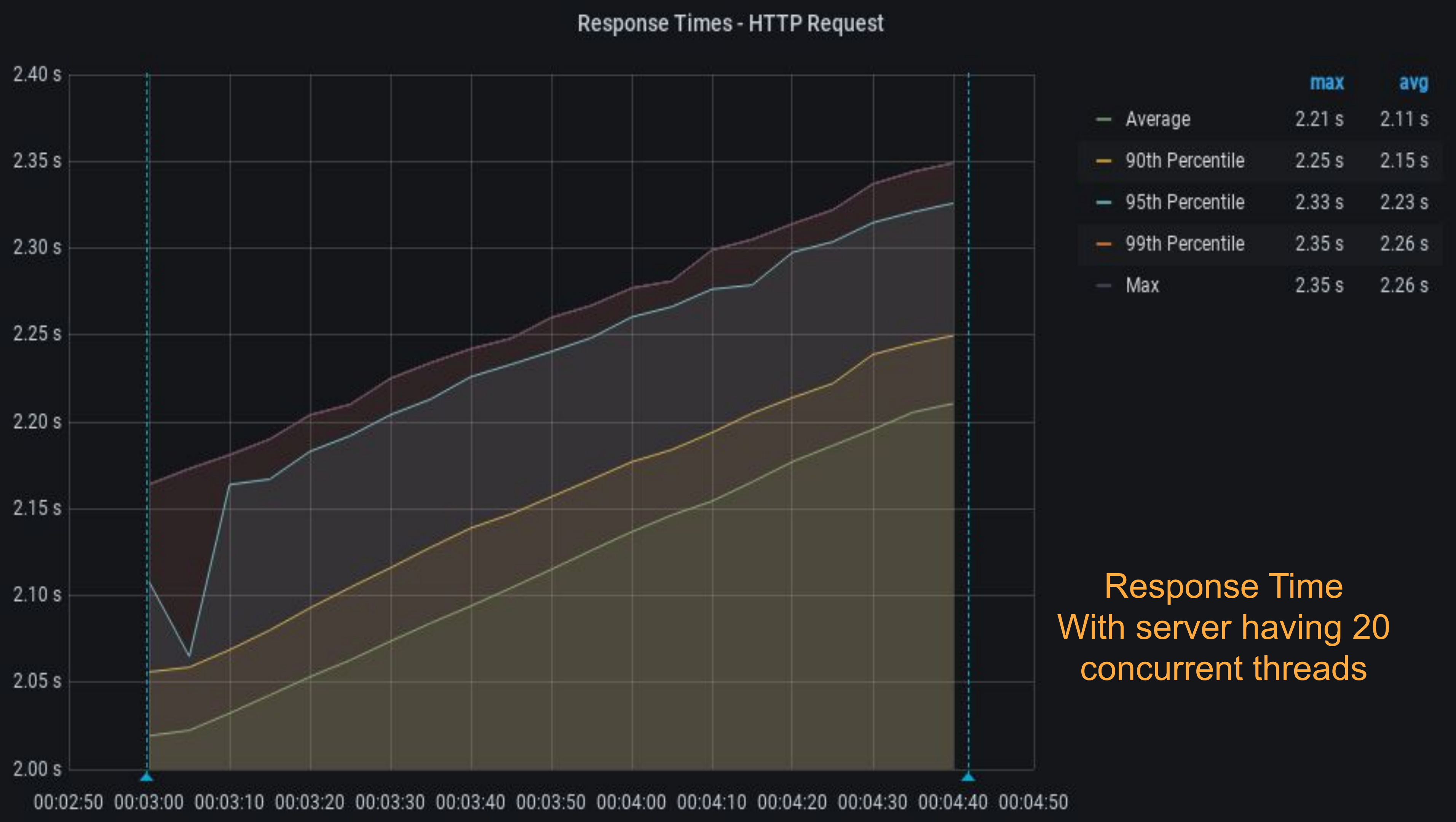
Network Traffic
With server having 10
concurrent threads



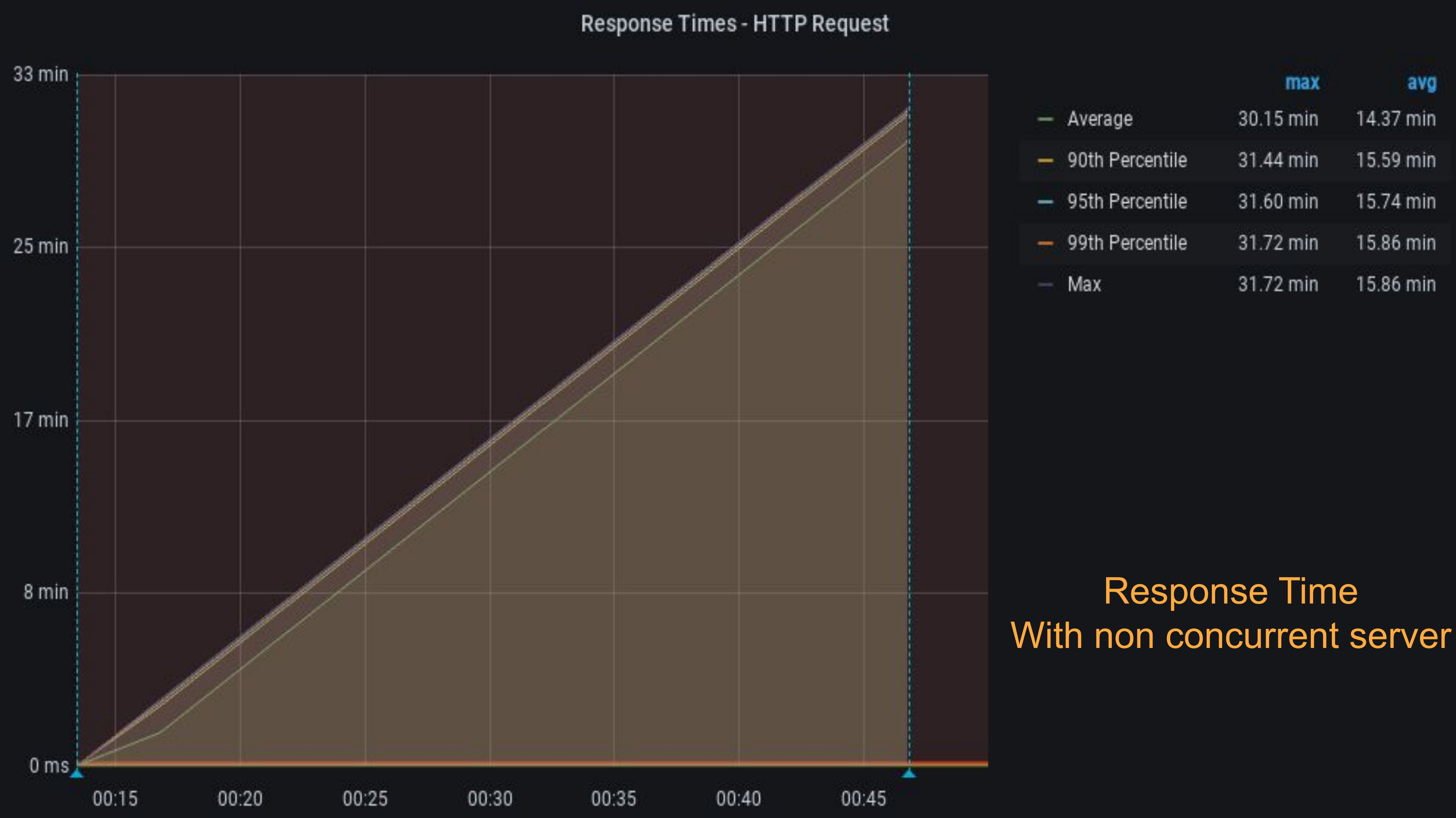
Network Traffic
With server having 20 concurrent threads



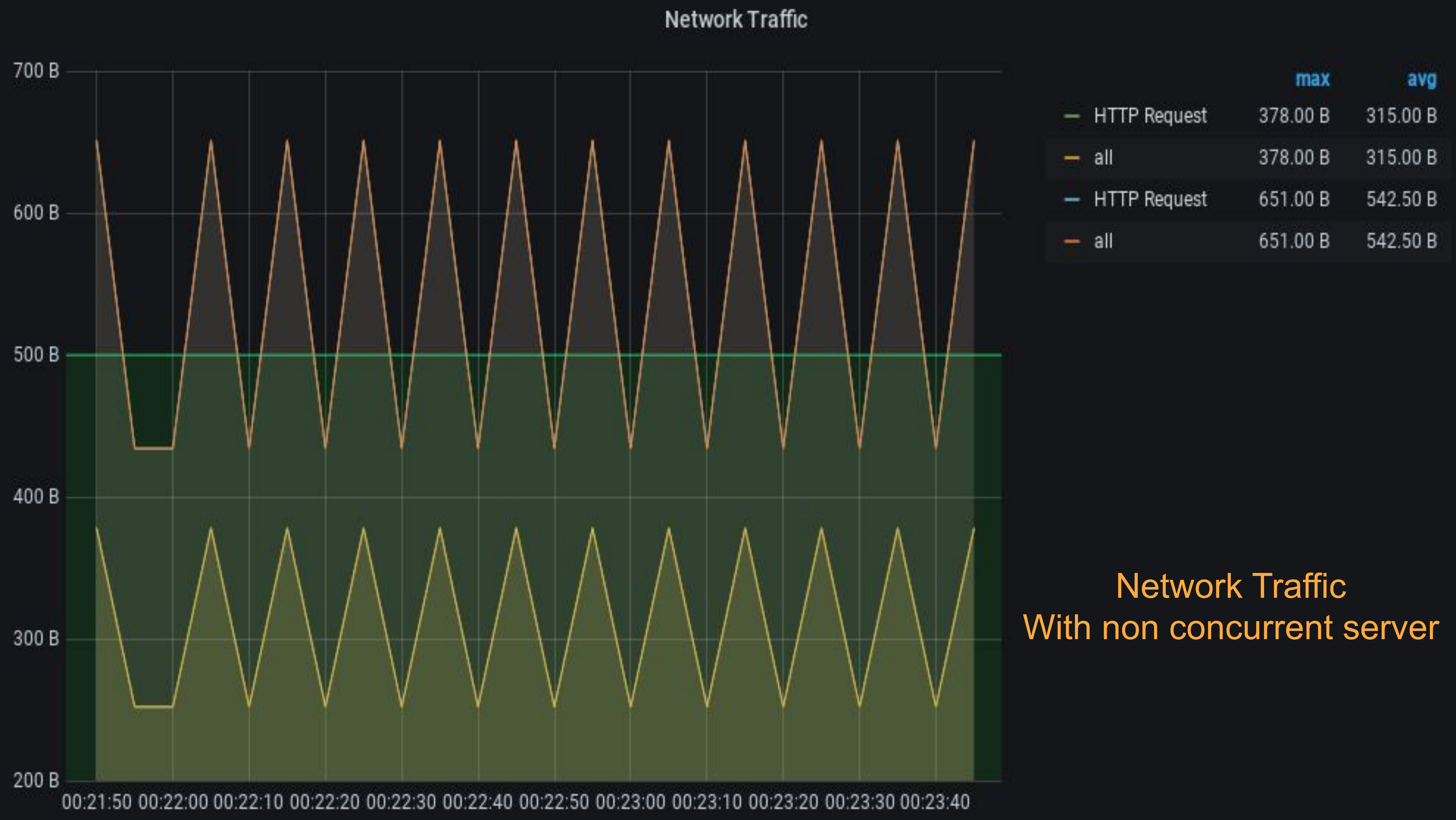
Response Time
With server having 10 concurrent threads



Response Time
With server having 20
concurrent threads



Response Time
With non concurrent server



Network Traffic
With non concurrent server