Performance, latency and scaling

Jarosław Rzeszótko
RTB House

Performance from the users perspective

Example user interaction with a web service (HTTP/1.0, no keepalive etc.):

- Type URL in the browser
- Resolve domain name in DNS
- Establish HTTP connection to main host
  - Establish TCP connection to host
  - Establish TLS connection
- Fetch HTML over the established connection
- Parse HTML, build DOM, start rendering
- Fetch assets: JS/CSS/images/fonts/etc.
- …

Performance from the users perspective

Example user interaction with a web service (HTTP/1.0, no keepalive etc.):

- …
- Fetch assets: JS/CSS/images/fonts/etc.
  - For each asset:
    - Resolve domain name in DNS
    - Establish TCP connection to host
    - Possibly establish TLS connection
  - …
- …

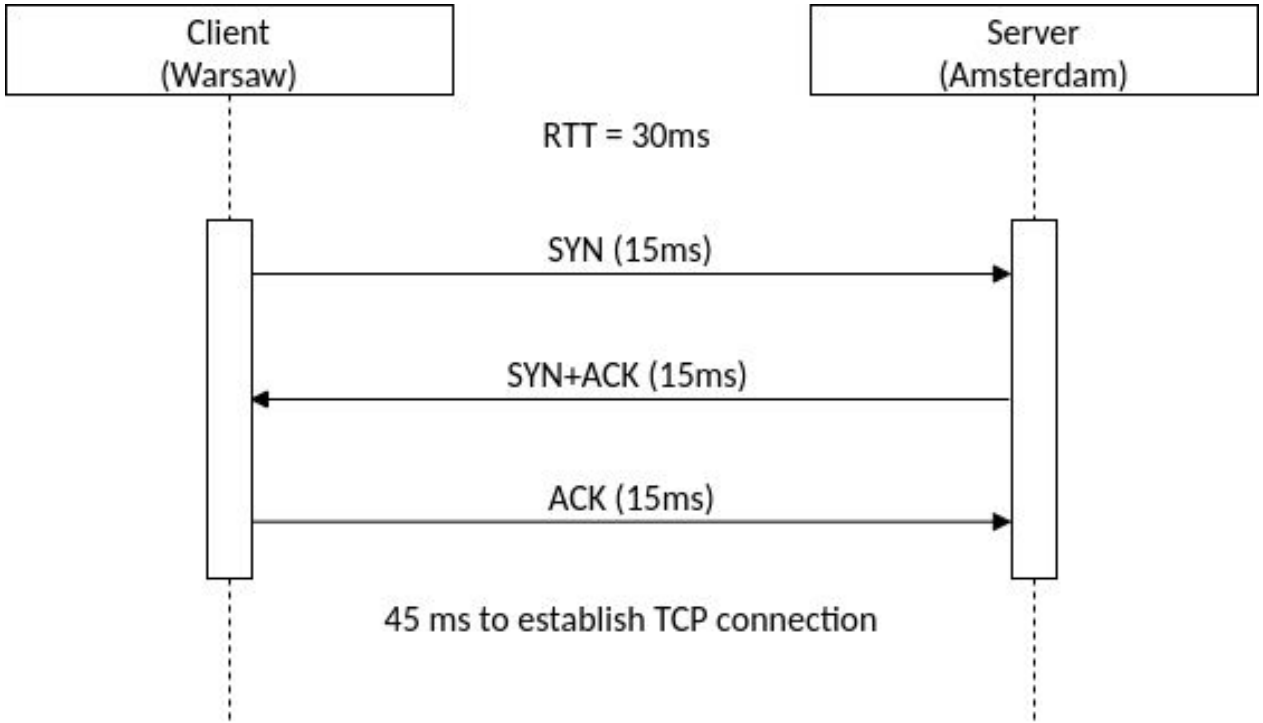
Problem: round trips

Round trip time / RTT / ping - time needed for any amount of data to flow from source to destination and back.

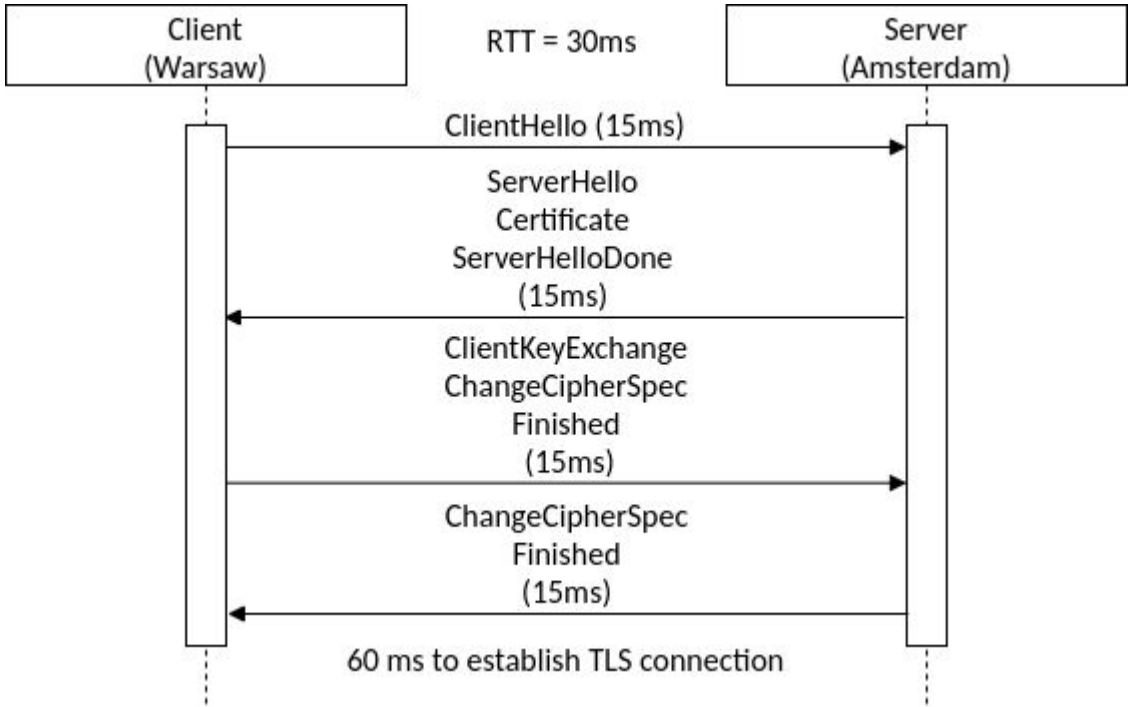
Can be measured with the ping command:

```
PING creativecdn.com (185.184.8.65) 56(84) bytes of data.
64 bytes from ip-185-184-8-65.rtbhouse.net (185.184.8.65): icmp_seq=1 ttl=56 time=32.5 ms
64 bytes from ip-185-184-8-65.rtbhouse.net (185.184.8.65): icmp_seq=2 ttl=56 time=32.5 ms
64 bytes from ip-185-184-8-65.rtbhouse.net (185.184.8.65): icmp_seq=3 ttl=56 time=35.0 ms
64 bytes from ip-185-184-8-65.rtbhouse.net (185.184.8.65): icmp_seq=4 ttl=56 time=35.7 ms
64 bytes from ip-185-184-8-65.rtbhouse.net (185.184.8.65): icmp_seq=5 ttl=56 time=33.1 ms
^C
--- creativecdn.com ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 32.467/33.740/35.652/1.335 ms
```

Problem: round trips



Problem: round trips



Problem: round trips

For two given geographic locations of client and server, round trip time has a physical lower limit given by the speed of light.

Problem: round trips

```
Speed of light in vacuum:           300 000 000 m/s
Speed of light in optical fiber:    200 000 000 m/s
Distance from Warsaw to Amsterdam:    1 100 000 m

1 100 000 m / 200 000 000 m/s = 0.0055 s = 5.5 ms

2 * 5.5 ms = 11 ms best-case round trip
```

https://tools.bunny.net/latency-test?query=creativecdn.com

Problem: round trips

TCP is not using full bandwidth immediately after handshake:

The OS sending data uses a **congestion control** algorithm that controls how much data is injected into the network.

Problem: round trips

- **Congestion window (cwnd)** on the sending side OS controls how many bytes can be in-flight.
- TCP starts in “slow start” (unfortunate name):

```
cwnd = 10*MSS =~ 15kB (on Linux, MSS - Maximum Segment Size)
cwnd += MSS after every ACK
```

- Exponential growth:

```
cwnd(t + RTT) = 2*cwnd(t)
```

- Continues until congestion is detected.

Problem: round trips

With 20ms RTT and no loss:

```
segments_acked(0ms)   = 0     cwnd(0ms)   = 10
segments_acked(20ms)  = 10    cwnd(20ms)  = 20
segments_acked(40ms)  = 30    cwnd(40ms)  = 40
segments_acked(60ms)  = 70    cwnd(60ms)  = 80
segments_acked(80ms)  = 150   cwnd(80ms)  = 160
segments_acked(100ms) = 310   cwnd(100ms) = 320
```

310*1.5kB = 465kB sent and ACKed after 100ms

Problem: head-of-line blocking in HTTP up to 1.1

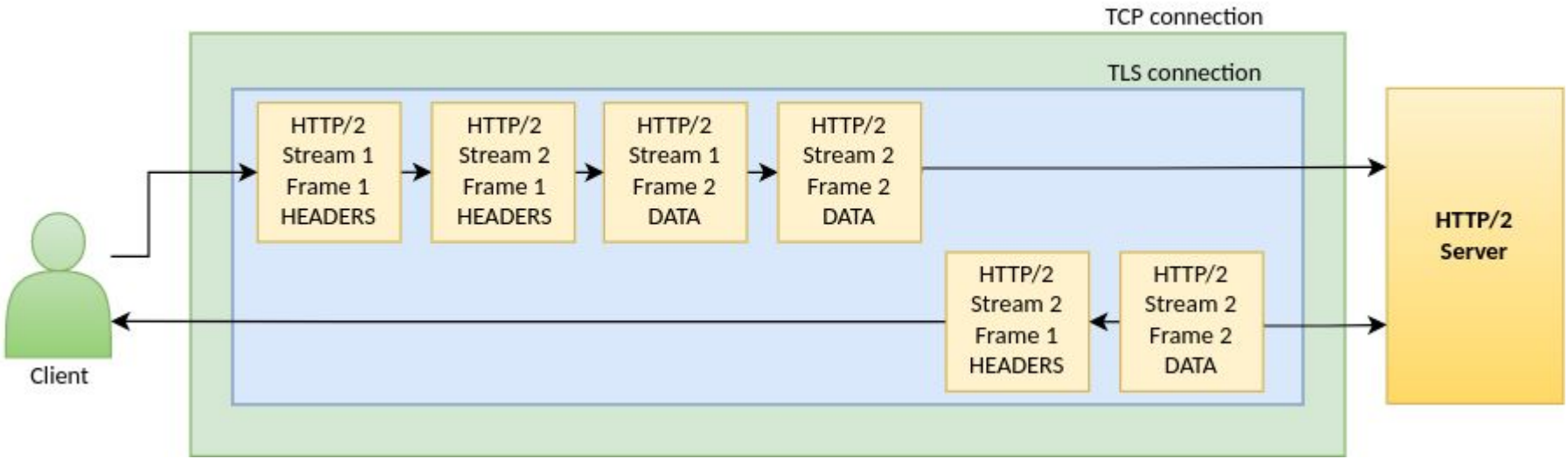
When a HTTP request is sent over a TCP connection (perhaps also TLS encrypted), second request can not be sent until the server processes the first request and response.

Head-of-line blocking: open more connections?

- Opening connections costs memory and CPU (with TLS)
  - In the client
  - In the server
  - In the infrastructure between the client and the server

HTTP/2

- Multiple interleaved streams:



Problem: head-of-line blocking in TCP

HTTP/2 still uses TCP, if the TCP stream is missing a byte somewhere in the stream, all HTTP/2 streams will be paused until that byte is retransmitted and received.

HTTP/3 & QUIC

- HTTP/3 is the application layer, QUIC is the transport layer
- QUIC is based on UDP
- QUIC does its own congestion control
- QUIC has a single handshake establishing an encrypted channel, instead of the two separate TCP and TLS handshakes in HTTP/2

Measuring end-to-end latency in the browser

Browsers expose APIs helpful for measuring latency:

- **Navigation Timing API**
  - For measuring root document load times
- **Resource Timing API**
  - For measuring load times of assets

Can collect data in client-side JS and send to some internal reporting API.

Performance from the service maintainers perspective

Service maintainer faces two basic performance-related questions:

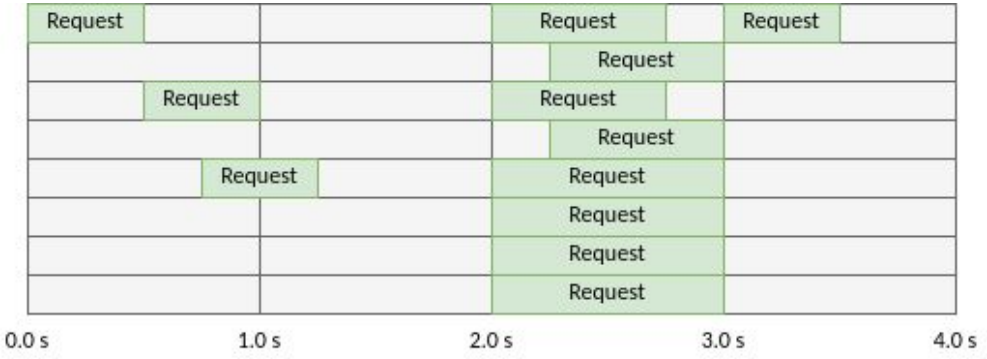
- What is the **maximum acceptable latency** for this service?
  - Determines the users experience
- What is the **maximum traffic** the service can handle, **while keeping latency acceptable**?
  - Determines the operating cost

Performance from the service maintainers perspective

To try to answer the basic questions we need some basic definitions:

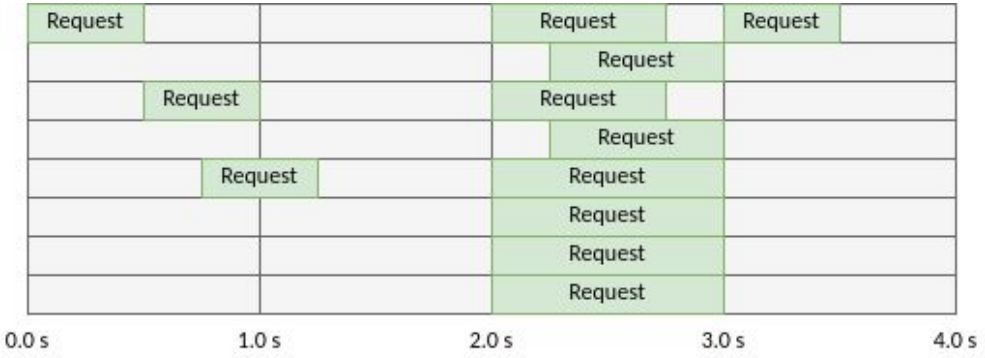
- Offered load = number of operations issued per unit of time
- Throughput = number of operations completed per unit of time
- Latency = time to complete a single operation
  (operation = HTTP request/RPC call/DB transaction/…)

Measuring offered load and throughput



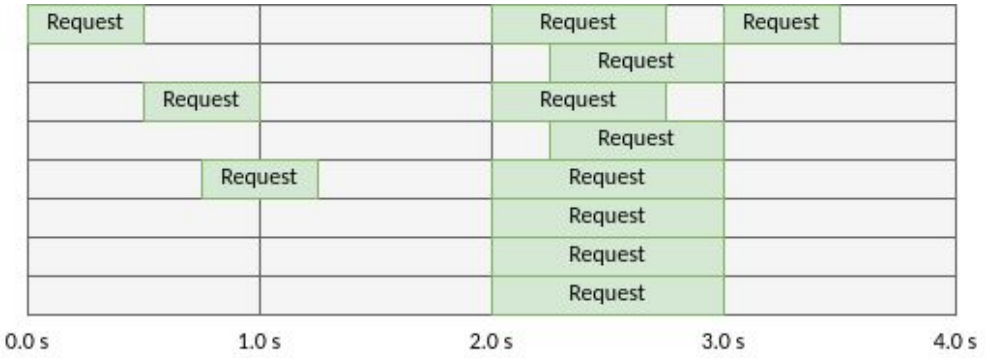
- Over the time period [0.0s, 1.0s]: offered load = 3 req/s, throughput = 2 req/s
- Over the time period [0.0s, 4.0s]: offered load = 12 req / 4s, throughput = 12 req / 4s
- Average over the time period [0.0s, 4.0s]: offered load = 3 req/s, throughput = 3 req / s

Measuring offered load and throughput



- To measure offered load:
  - Increment counter each time a request arrives
  - Each T seconds store the value of the counter

Measuring offered load and throughput



- To measure throughput:
  - Increment counter each time a response has been sent
  - Each T seconds store the value of the counter

Measuring offered load and throughput

As a result servers produce streams of per-second counts as metrics:

Server 1:

```
offered load: [12:00:00: 1 req started,   12:00:01: 5 req started, …]
throughput:   [12:00:00: 1 req completed, 12:00:01: 3 req completed, …]
```

Server 2:

```
offered load: [12:00:00: 2 req started,   12:00:01: 5 req started, …]
throughput:   [12:00:00: 2 req completed, 12:00:01: 5 req completed, …]
```

Measuring offered load and throughput

This stream of per-second counts can be stored in a time-series database like Prometheus or InfluxDB, and then queried in different ways:

- What was the average request rate/s between 12:00 and 12:01 across all servers?
- What was the throughput of the slowest server between 12:00 and 13:00?
- …

Measuring offered load and throughput

Why is the distinction between offered load and throughput important?

Imagine you are monitoring a server that suddenly started suffering from increased response latency.

The first thing to do is to find out if:

A. the system is overloaded because of increased traffic (offered load), or
B. the system has slowed down (e.g. due to a code change, config change, etc.) ?

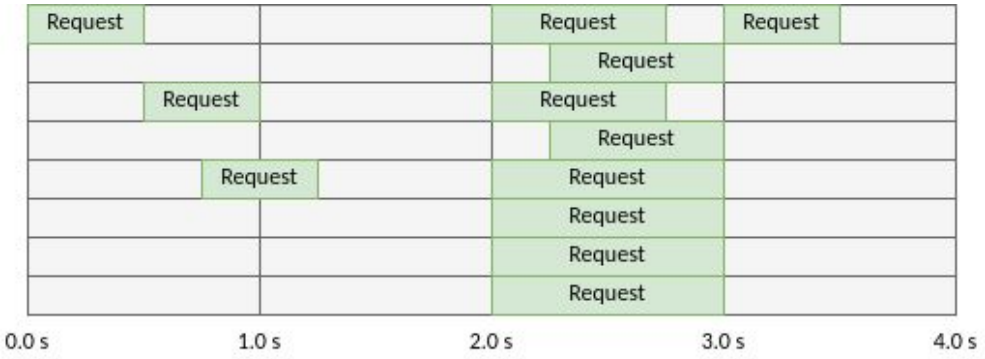
Measuring offered load and throughput

- If the system is overloaded by increased traffic:
  - Offered load has increased
  - Throughput could have stayed the same or even decreased!
  - As offered load keeps increasing throughput:
    - increases slower and slower
    - stops increasing at all
    - might start decreasing

- If the system has slowed down:
  - Offered load has stayed the same (e.g. compared to yesterday)
  - Throughput has decreased

If you only monitor throughput, you can’t tell if the system is overloaded or slowed down!

Measuring latency

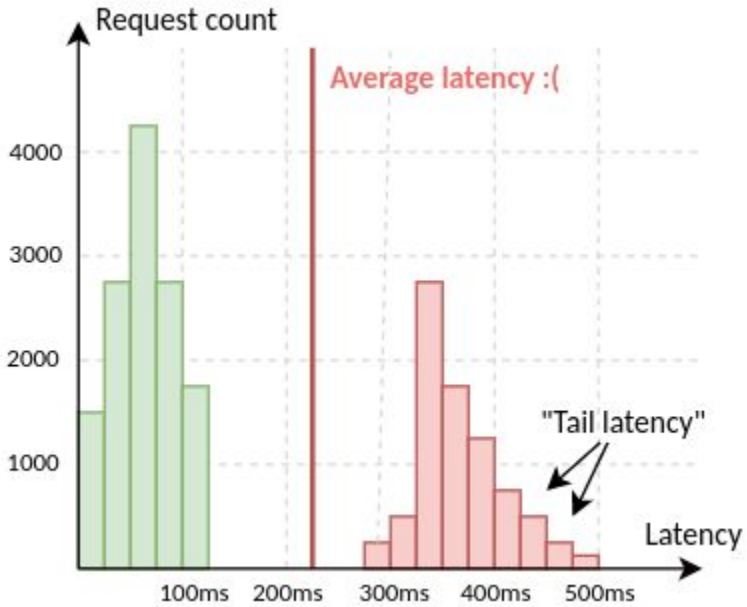


- To measure latency
  - Store request durations
  - Each millisecond/second/minute store the average latency?
  - ???

Measuring latency

Why is average latency confusing?

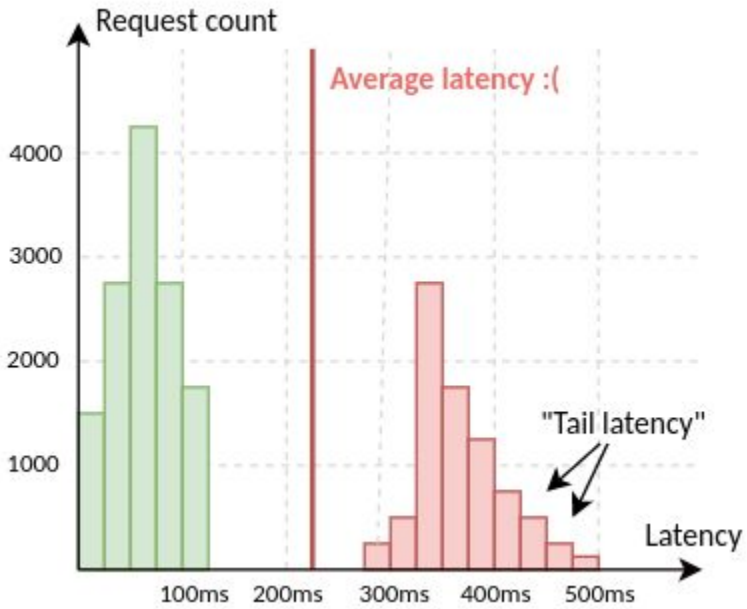
Measuring latency



Why is average latency confusing?

- Latency distributions rarely are unimodal or symmetric
- Green can be fast-path, red can be slow-path
- Green can be hot cache, red can be cold cache
- Green can be requests without GC pause, red can be requests with GC pause
- Several of the above combined together can result in more clusters than just two

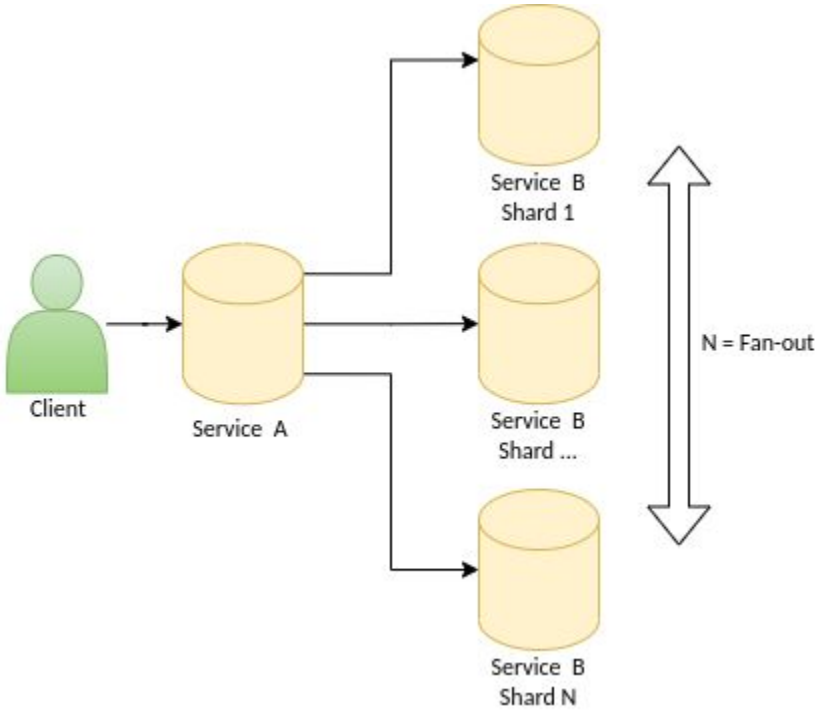
Measuring latency



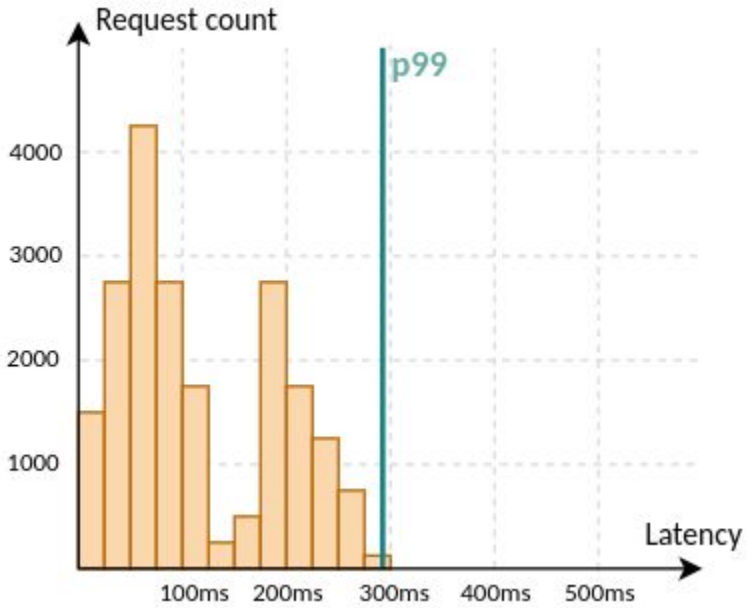
Why is average latency confusing?

- User action might trigger several parallel requests and from the users perspective be complete only when all requests complete

Measuring latency



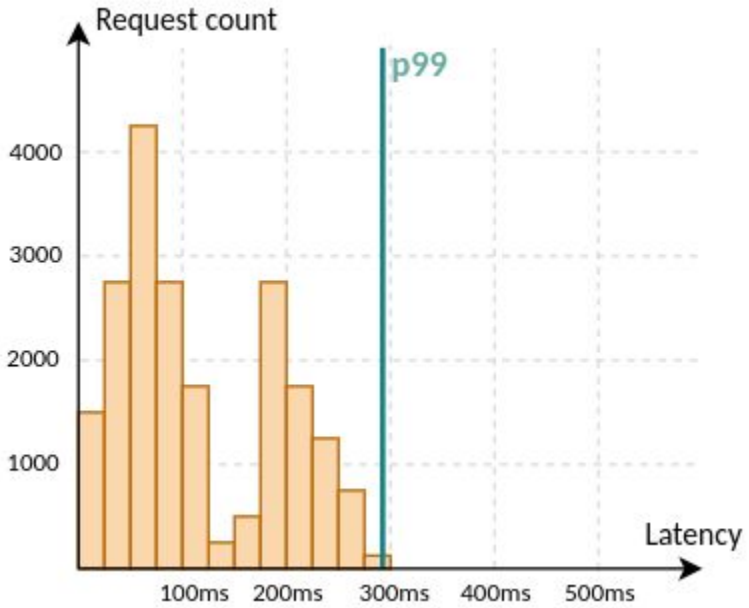
Measuring latency



What do instead:

- Want to cap high percentiles: p95/p99/p999/… at X ms so that user is unlikely to experience latency > Xms
- p95: X such that 95% data points are <= X
- p99: X such that 99% data points are <= X
- …

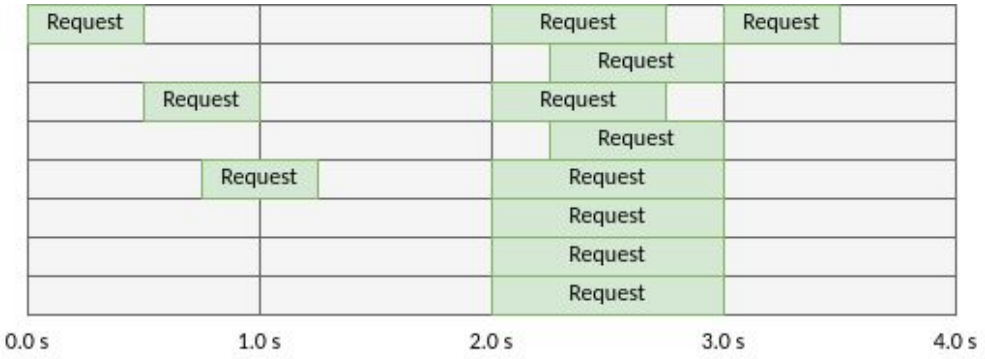
Measuring latency



What percentile to use?

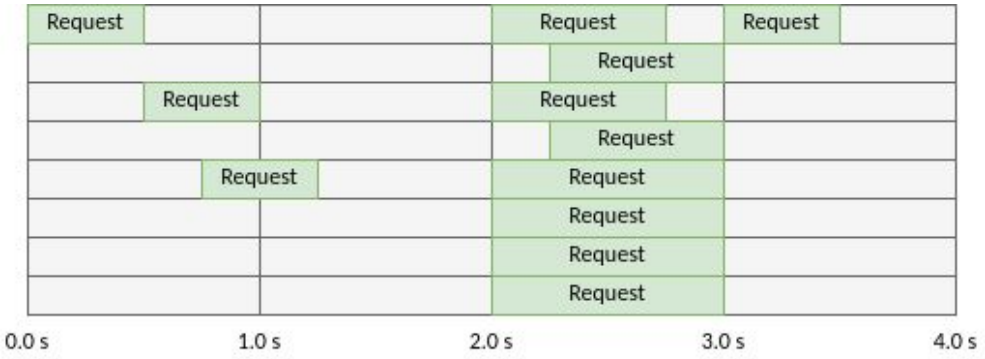
- No golden rule, but…
- the higher the fan-out the higher the percentile
- on the other hand it is much harder to lower the p999 to 100ms than to lower the p99 to 100ms
- there are dozens of strange reasons requests can occasionally slow down

Measuring latency



- To measure latency
  - Store request durations
  - Each millisecond/second/minute store p95/p99/p999/p9999 latency?
- What is wrong with this?
  - Hint: what if this is one server from many?

Measuring latency



- To measure latency
  - Define histogram buckets, for example [0ms, 100ms], [100ms, 200ms], …
  - Every millisecond/second/minute store the count of requests in each histogram bucket
  - Can aggregate both over time and over different servers
  - Can calculate approximate percentiles

Performance from the service maintainers perspective

Service maintainer faces two basic performance-related questions:

- What is the **maximum acceptable latency** for this service?

- What is the **maximum throughput** the service can achieve **while keeping latency acceptable**?

Performance from the service maintainers perspective

- What is the **maximum acceptable latency** for this service?
    - Define SLA for service performance: p95/p99/p999 <= X ms

Performance from the service maintainers perspective

- What is the **maximum throughput** the service can achieve **while keeping latency acceptable**?
    - Have to run load tests benchmarks

Benchmarking servers

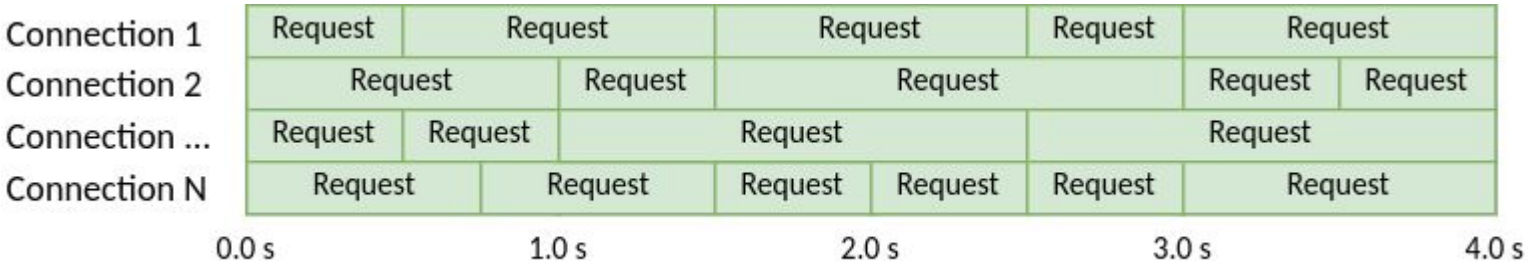
How would you benchmark an HTTP server?

Benchmarking servers

Simplest way implemented by most basic benchmarking tools:

1. Setup some (initially small) number of connections N
2. Send a request over each connection
3. When a request on some connection completes,
   send a new request on the same connection
4. Continue sending requests this way for T seconds
5. After T seconds elapse, report average number of responses/s and latency statistics
6. Increase N and repeat

Benchmarking servers



Benchmarking servers

Simplest way implemented by most basic benchmarking tools:

Test is completed when the test workload causes a violation of the SLA.

Previous throughput value (from the test run which did not yet violate the latency SLA)
is the **maximum throughput.**

This is called a **closed system model,** because it simulates having a fixed number of users with no user arrivals and no user departures.

Ideal linear scaling

Assume:

- System has N processors
- C identical tasks are always available, C <= N
  (C is the concurrency level, or load in the closed system model)
- One task is fully done by a single processor
  (one thread handles one request from start to finish)

We test task completions / sec and mean/p50/p90/p95/… task latency,
as a function of increasing C:

1 <= C <= N

Ideal linear scaling

With linear scaling:

- Latency L is constant and independent of the concurrency level C
- Throughput T = C * (1/L)

Reasons for non-linear scaling: explicit synchronization

Reasons for non-linear scaling: CPU caches & Hyper-Threading

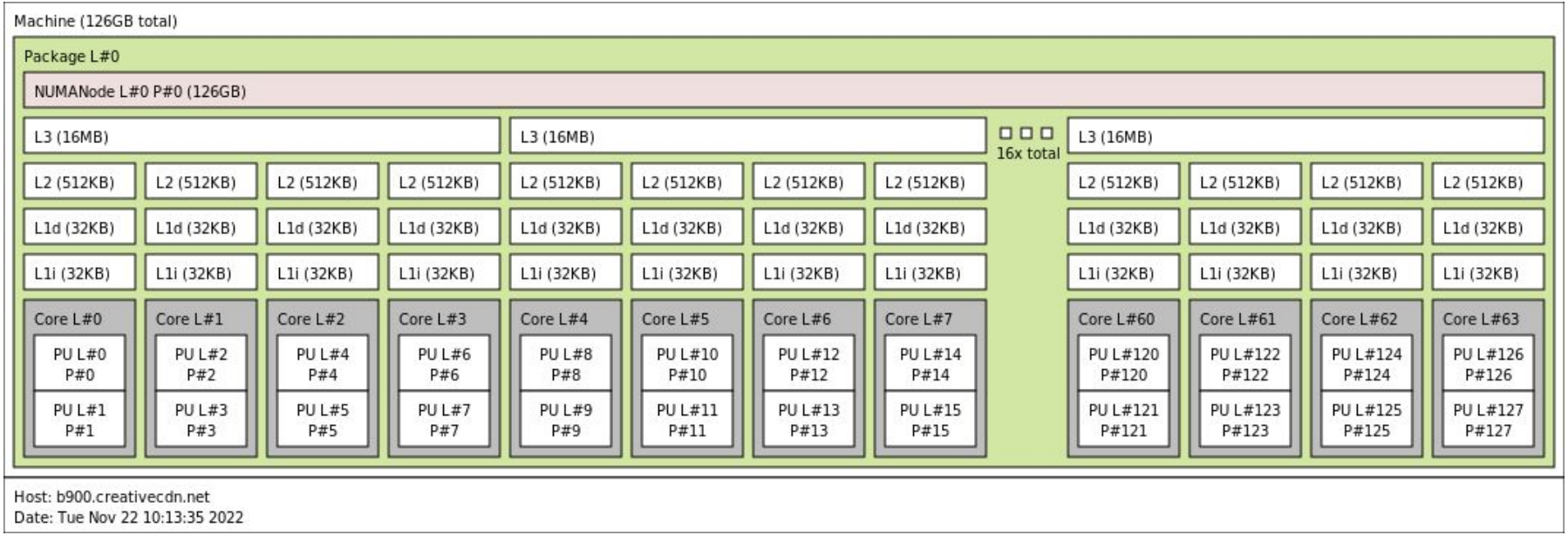
- Over the last 40 years, CPUs were getting faster at a faster rate than memory
- CPUs are now much faster than memory
- CPU design has to deal with the memory bottleneck
- CPU cache hierarchy and Hyper-Threading both try to address the bottleneck

Reasons for non-linear scaling: CPU caches & Hyper-Threading

- CPU cores visible in the OS are NOT independent execution units
- Last Level cache (L3 cache) is typically shared between many physical cores
- With Hyper-Threading, single physical core is visible as two logical cores in the OS

Reasons for non-linear scaling: CPU caches & Hyper-Threading

Output of hwloc command executed on a server with AMD EPYC 7702P CPU:



Reasons for non-linear scaling: Hyper-Threading & CPU caches

Idea behind caching:

- In hardware design so far, larger memory banks imply higher memory access latency
- Exploit **locality of reference** to give illusion of memory that is both fast and large
- **Spatial locality**: If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.
- **Temporal locality:** If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near

Reasons for non-linear scaling: Hyper-Threading & CPU caches

Idea behind caching:

- Hierarchy of caches
- Per-core very fast & very small L1 cache: 32kB, 1.18ns / 4 cycles access latency
- Per-core fast & small L2 cache: 512kB, 3.86 ns / 13 cycles access latency
- Larger and slower L3 cache shared between cores: 16MB / core complex, 10.27 ns / 34 cycles access latency

Reasons for non-linear scaling: Hyper-Threading & CPU caches

Downsides of caching:

- There can be a very sharp performance downgrade each time when the workload stops fitting in given level of the cache
- Programming practices like heavy reliance on pointers and non-contiguous memory can defeat caching and drastically reduce performance
- Cores sharing L3 cache can not be treated as independent

Reasons for non-linear scaling: Hyper-Threading & CPU caches

Idea behind hyper-threading:

- A physical CPU core can spend a large percentage of time stalled waiting for memory
- Present a single physical CPU core as two or more logical cores to the OS
- OS can assign two different threads to two logical cores backed by the same physical core
- While the execution of the first thread is stalled waiting for memory, second thread can be executed

Reasons for non-linear scaling: Hyper-Threading & CPU caches

Downsides of hyper-threading:

- The two logical cores present in the OS sharing a physical core are not independent

- There is a performance downgrade when server utilization cross the threshold where the OS scheduler can dedicated a single physical core (almost) exclusively to a single thread

Issues with the closed-system model: coordinated omission

See: "How NOT to Measure Latency" by Gil Tene
https://www.youtube.com/watch?v=lJ8ydIuPFeU

Issues with the closed-system model: coordinated omission

Problem: less queuing than real life

- Test caps the number of concurrently active connections to N
  -> there are never more than N requests in the middle of processing

- Based on the completed requests/s reports average throughput X ops/s

- When you observe an average offered load of X requests/s over a minute in real life, it might be that at T=0s, X requests arrived all at once, X >> N

Open-system model

1. Assume a request rate R and a probability distribution of request interarrival times
2. Generate a plan up-front: timestamps at which requests should be sent
3. Grow and shrink the connection pool and/or thread pool of the benchmarking tool as needed
4. Increase R
5. Repeat 1-4 until latency SLA is violated

This is called a **open system model**.

Queuing

When:

- offered load > throughput

for some prolonged period of time, it means that either:

A. Requests are getting dropped
B. Requests are getting queued

Queueing example: HTTP server with new thread per request

- HTTP server spawns new thread for each incoming request
- With enough in-flight HTTP requests there will be more server threads than CPUs
- Some of the threads will be executing, others will wait in the OS scheduler queue

Queuing example: HTTP server with explicit queue

- HTTP server has N worker threads for processing incoming requests
- Incoming requests are placed into a concurrent queue
- Each of the worker threads works in a loop:
  - ```
    while (request = queue.poll()) {
        request.process()
    }
    ```
- If a thread is idle when request arrives, handoff is nearly instant
- Otherwise request typically has to wait until it reaches the front of the queue (FIFO)
    ```

Queuing

Real traffic typically is bursty

Queuing smoothes out the bursts

Queuing increases utilization at the cost of latency

Queuing can not rescue the service from not having enough capacity

Unlimited queuing can be harmful

Queuing

When:

- offered load > throughput
- requests are getting queued
- there is no limit to the queue length

Then:

- The queue with continue to grow, consuming resources (memory, sockets, etc.)
- Latency will continue to grow

Might eventually break in various ways, e.g. run out of memory and get killed by the OS

Load shedding

Example: incoming request/s exceed completed requests/s by 5% over an hour

What to do?

Possibilities:

A. Queue the excess 5% - latency will increase over the course of the hour, might run out of memory, out of sockets, etc.
B. Drop the excess 5% - e.g. return empty responses, serve the remaining traffic in reasonable time
C. Serve some % of requests using a faster code path - degrade service quality to improve throughput

Load shedding

Some way to decide when to start shedding the load is needed.

Possibilities:

A. Monitor average response latency over last X seconds
B. Monitor average thread pool utilization over last X seconds (sample every 100ms, keep last 10 samples, use average of 10 samples to decide whether to shed load or not)

Summary

- Monitor offered load to tell apart system overload from system slowdown
- Latency increases as offered load increases,
  often even at low/moderate CPU/disk/etc utilization
- Tail latency matters, in particular for services with large fan out
  the larger the fanout, the larger part of the “tail” matters

Summary

- Queues are everywhere
- Overload = offered load > maximum possible throughput
- On overload, requests either are dropped or queued
- Queues achieve higher utilization at the cost of higher latency, sensible for short bursts of increased traffic

Summary

- If overload persists for longer time and requests get queued, latency goes to infinity as time goes to infinity

- Services should implement a deliberate strategy of handling overload, service maintainers need to be conscious of the strategy the service implements