Server Performance

Practical Distributed Systems 2022: Lecture 2

Jarosław Rzeszótko
RTB House

Users experience using an online service

- Type URL
- Resolve domain
- TLS handshake
- Fetch HTML
- Fetch assets
- Click button
- **Do operation**
- …

Performance from the service maintainers perspective

Service maintainer faces two basic performance-related questions:

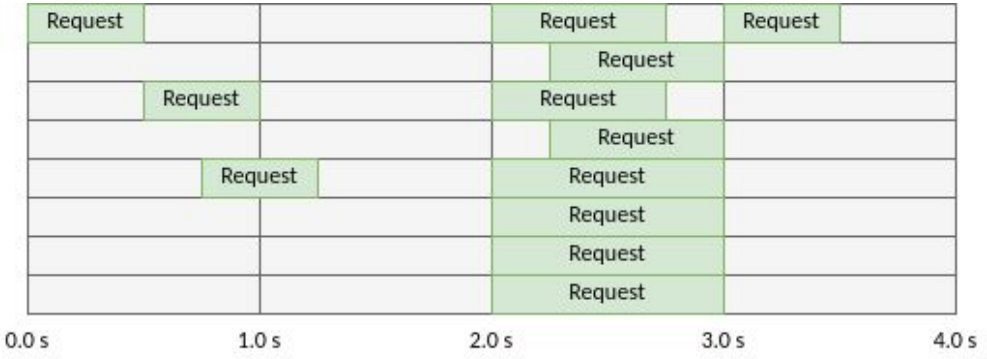
- What is the **maximum acceptable latency** for this service?
  - Determines the users experience
- What is the **maximum traffic** the service can handle, **while keeping latency acceptable**?
  - Determines the operating cost

Performance from the service maintainers perspective

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

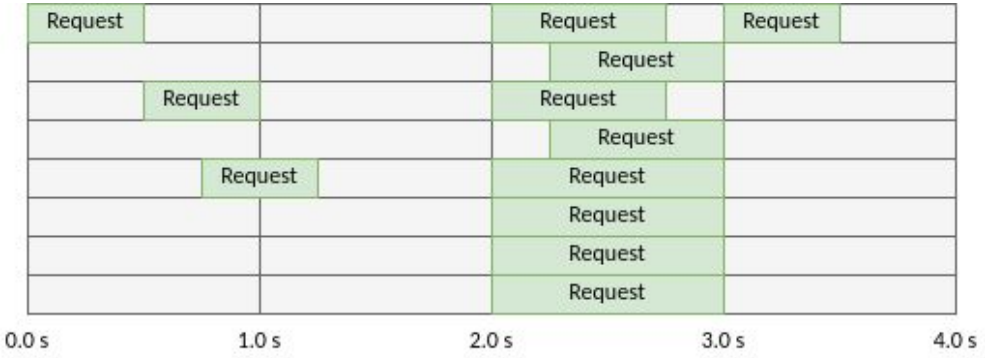
- Offerred load = number of operations issued per unit of time
- Throughput = number of operations completed per unit of time
- Latency = time to complete a single operation
  (operation = HTTP request/RPC call/DB transaction/…)

Measuring offered load and throughput



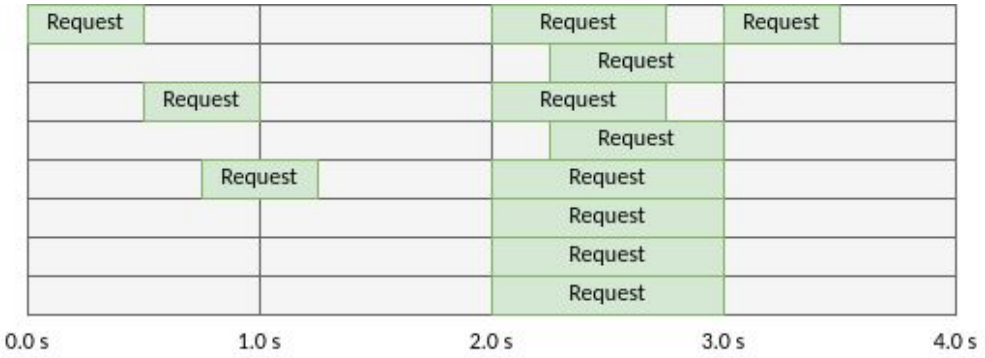
- Over the time period [0.0s, 1.0s]: offered load = 3 req/s, throughput = 2 req/s
- Over the time period [0.0s, 4.0s]: offered load = 12 req / 4s, throughput = 12 req / 4s
- Average over the time period [0.0s, 4.0s]: offered load = 3 req/s, throughput = 3 req / s

Measuring offered load and throughput



- To measure offerred 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.) ?

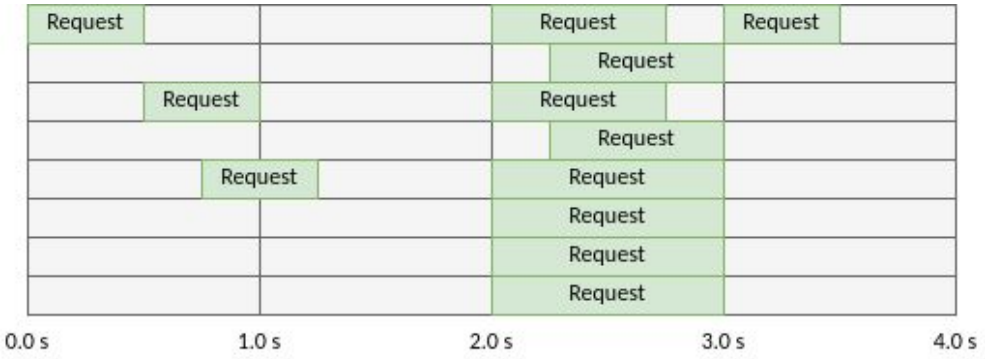
Measuring offered load and throughput

- 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

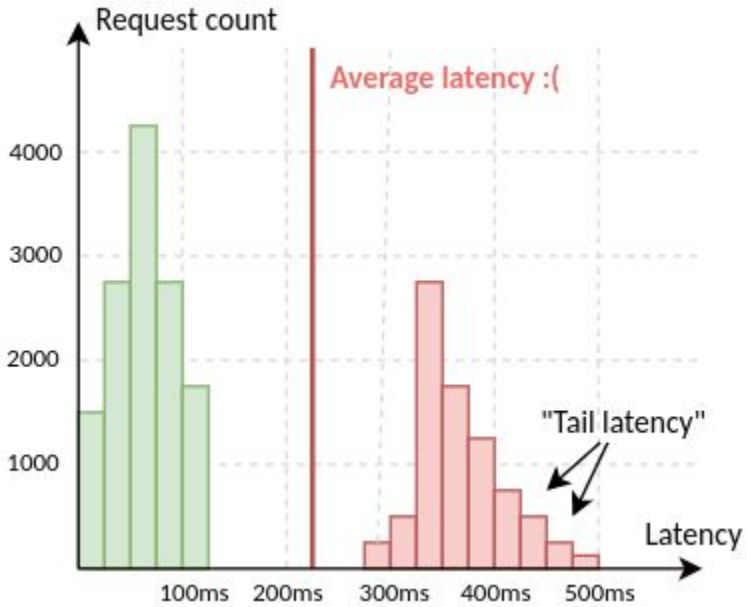


- To measure latency
  - Store request durations
  - Each millisecond/second/minute store the average latency?
  - ???

Measuring latency

Why is average latency confusing?

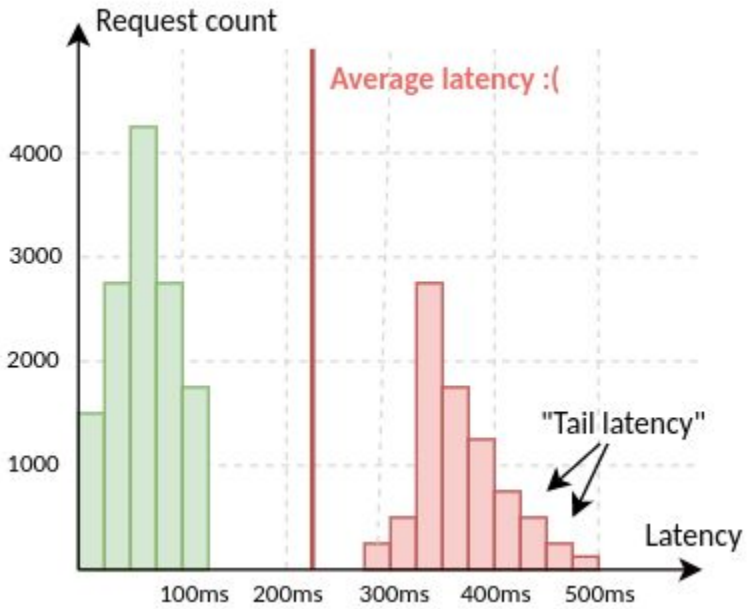
Measuring latency



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

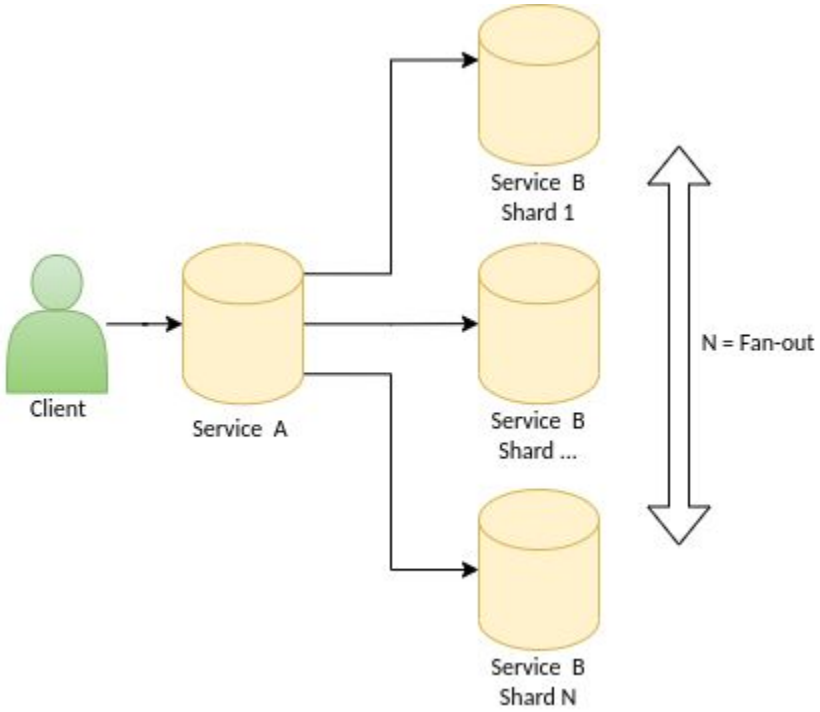
Measuring latency



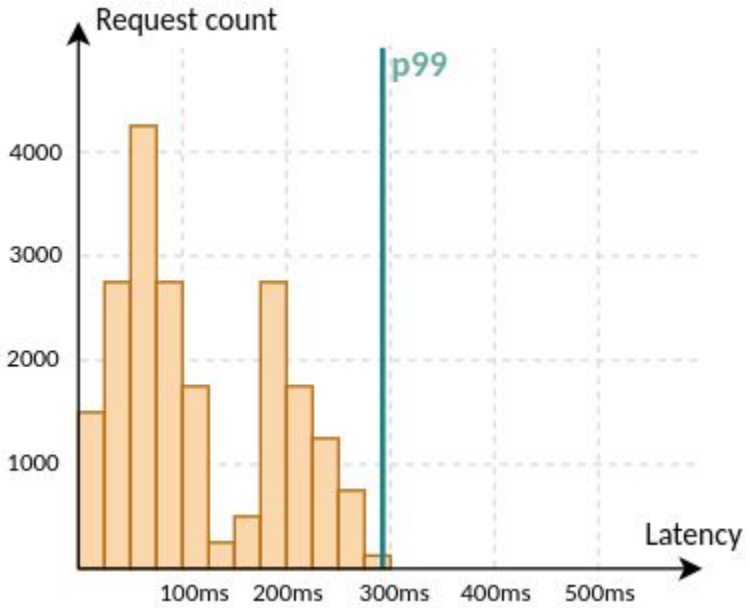
Why is average latency confusing?

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

Measuring latency



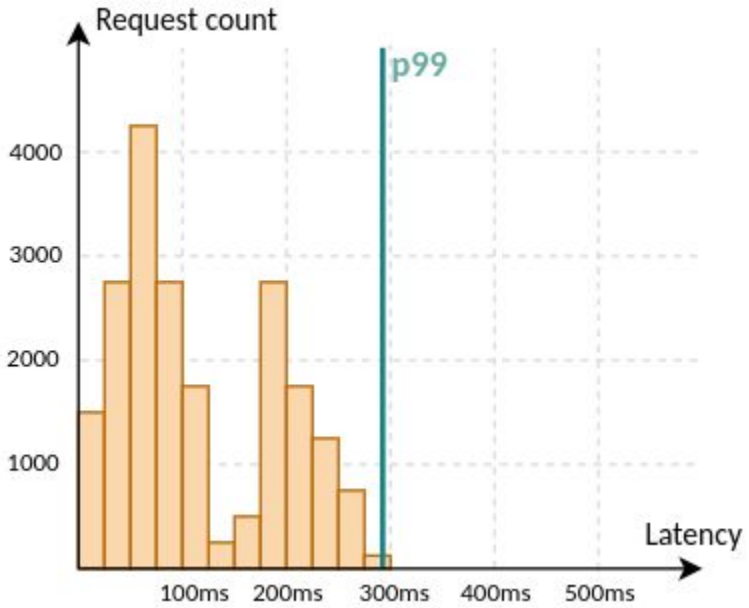
Measuring latency



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
- …

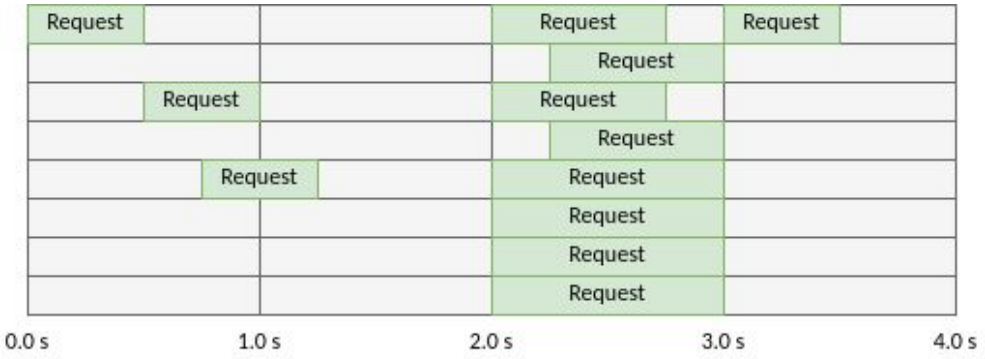
Measuring latency



What percentile to use?

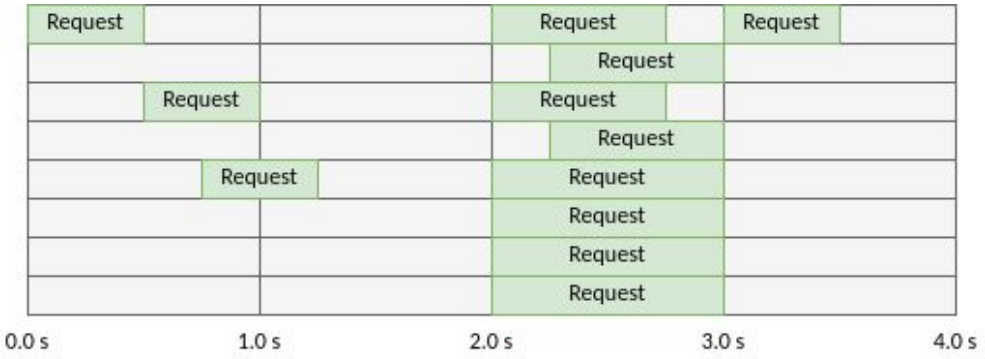
- No golden rule, but…
- the higher the fan-out the higher the percentile
- on the other hand it is much harder to lower the p999 to 100ms than to lower the p99 to 100ms
- there are dozens of strange reasons requests can occasionally slow down

Measuring latency



- To measure latency
  - Store request durations
  - Each milisecond/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 milisecond/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

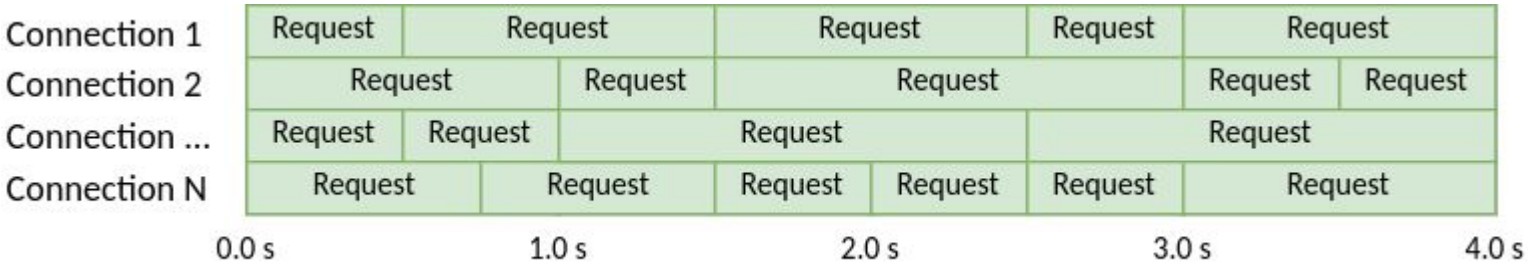
- What is the **maximum throughput** the service can achieve **while keeping latency acceptable**?
    - Have to run load tests benchmarks

Benchmarking servers

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.

Benchmarking servers

Simplest way implemented by most basic benchmarking tools:

What is wrong with it?

Benchmarking servers

Problem: coordinated omission

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

Benchmarking servers

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

Benchmarking servers

Another way:

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**.

Relation between offered load and latency

Assume:

- There are N fully independent processors
- It takes T seconds for the processor to complete some computation of interest

Then, provided all processors always have work available, throughput is simply:

N * 1s / T

Relation between offered load and latency

In reality:

- Some stages of processing can be completed with parallelism P1, others with parallelism P2, …
- Requests arrive at different time intervals
  - Sometimes there is more parallelism available than there is work items available
  - Sometimes there are more work items than available parallelism and work items wait in queue
- Processors might need to synchronize
- …

Queuing

When:

- offered load > throughput

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

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

Queuing 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?

Possiblities:

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 detect when to start to shed the load is needed.

Possiblities:

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

Resources

Tail latency in large distributed systems:

- **The Tail at Scale** by Jeff Dean and Luiz André Barroso from Google
  Paper about the impact of tail latency on large distributed systems and about techniques of mitigating it
  https://research.google/pubs/pub40801/