What’s the cost of a millisecond?

Avishai Ish-Shalom (@nukemberg)
ScyllaDB what?

+ The Real-Time Big Data Database
+ Drop-in replacement for Apache Cassandra and Amazon DynamoDB
+ 10X the performance & low tail latency
+ Open Source, Enterprise and Cloud options
+ Founded by the creators of KVM hypervisor
+ HQs: Palo Alto, CA, USA; Herzelia, Israel; Warsaw, Poland
ScyllaDB

+ The Real-Time Big Data Database
+ Drop-in replacement for Apache Cassandra and Amazon DynamoDB
+ 10X the performance
+ Open Source, Enterprise and Cloud options
+ Founded by the creators of KVM hypervisor
+ HQs: Palo Alto, CA, USA; Herzelia, Israel; Warsaw, Poland
The curse of latency amplification
5ms + 5ms + 5ms + 1ms = 16ms, right?
Once more, with amplification
Oh sh!t

- Every queue amplifies latency
- A timeout has a large penalty
- Retry has a penalty

And it all compounds. And transactions impact each other
Queueing theory crash course
Incoming work

Queue

Service center

Wait time (queueuing)

Service time (actual work)

Observed latency = wait time + service time
Head of line blocking

+ When some task takes longer, service center is “blocked”
+ Other tasks in the queue are blocked by the “head of line”
+ A single slow task will cause a **bunch of other tasks** to wait
  + Bad news for latency high percentiles
Latency (and queue size) rises to infinity as utilization approaches 1
Decent latency -> over capacity

\[ Q \propto \frac{\rho}{1 - \rho} \]

\( \rho = \text{arrival rate} / \text{service rate} = \text{utilization} \)

\( Q = \text{Queue length} \)

http://queuemulator.gh.scylladb.com/
Kingman formula

\[ \mathbb{E}[W_q] \approx \left( \frac{\rho}{1 - \rho} \right) \left( \frac{c_a^2 + c_s^2}{2} \right) \tau \]

- The higher the variance, the worse the latency/utilization curve gets
- On both service rate and arrival rate
- high variance \(\Rightarrow\) run at low utilization

Oh and btw your percentile curve is worse too*
Tasks should be independent, but...

+ Shared resources have queues
  + Disks, CPUs, Thread pools, Load balancers, connection pools, DB locks, sockets...
+ Head-of-line blocking → cross task interaction
  + Slow tasks raise latency of unrelated tasks
  + Arrival spikes
+ High variance service makes this worse
+ Parallel queues are less susceptible, but are less efficient
  + Some queues will be starved → lower utilization, throughput
Executive summary

+ High utilization → high latency
  + Non-linear!
+ High variance → high latency
+ Shared queues* → higher throughput, lower latency
+ **Never** use unlimited queues

* For identical service centers
Amplification sources
Queueing

+ Queues are everywhere
  + LB, locks, resource pools, sockets, event loop... and ofc, queues
+ Non linear rise in latency when load rises
  + Very problematic when running near capacity limits
+ Often not monitored
Timeouts

Break when something takes too long (or won’t complete)

+ Timeout values often arbitrary
+ Often wayyyy too long
  + Example: HikariCP acquire() min timeout = 250ms (!!!)
+ Often blocking service centers/other resources

*Power of ten syndrome: 100 is a bogus number*
Retry

If at first you don’t succeed, try again!

- But this takes even more time
- Especially if you have long timeouts
- How many retries?
  - Do they all have the same timeout?
Fork/Join

Spawn multiple parallel tasks, wait for all

+ Blocked until last task is complete
+ High probability of hitting at least 1 high percentile
Every cross service call has amplification

And they compound: $A_1 \times A_2 \times A_3 \ldots$

Need to wait for all - in sequence; Like fork/join only worse
  
  Growing probability of at least one p95/failure/timeout/overload...
Combating latency amplification
Proper timeouts

For the love of god, measure!

+ Use latency percentiles/histogram to determine correct timeouts
  + failure rate ↔ max latency
+ Compare with failure cost (e.g. reconnect)
+ Timeouts don’t have to be static!
  + E.g. \( \text{timeout} = \text{Min}(P999[\text{last 5m}], 300\text{ms}) \)
  + Lower timeout on high load
Timeout budget

- Global latency budget for request
  - Pass on request context
- Decrement actual processing on every stage
- Timeout = min(remaining budget, local timeout)
- Preemptive abort: fail if not enough budget

Very useful with microservices, but needs protocol support

<table>
<thead>
<tr>
<th>Service</th>
<th>Budget</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service 1</td>
<td>500ms</td>
<td>123ms</td>
</tr>
<tr>
<td>Service 2</td>
<td>377ms</td>
<td>72ms</td>
</tr>
<tr>
<td>Service 3</td>
<td>305ms</td>
<td>287ms</td>
</tr>
<tr>
<td>Service 4</td>
<td>18ms</td>
<td>reject</td>
</tr>
</tbody>
</table>
Parallel dispatch

+ Double dispatch: ask twice, wait for first answer
  + But also costs twice
+ Speculative execution: get data before you need it
+ Branch prediction: get data you *might* need
+ Harvest/yield: ask multiple shards, replicas; proceed with the answers you got within the timeout
Smarter retries

+ Speculative retries: retry even without failure, wait for first answer
  + Cheaper than double dispatch, very effective
  + Your API is idempotent, right?
+ Second retry can have shorter timeout (use the budget, Luke!)
+ Probabilistic retries: why retry if you can’t succeed
Capacity/latency management

Overloaded service centers will have higher latency amplification

- Limit concurrency according to Little’s law
- Cap queue lengths
- Run slower service with lower utilization
- Run high variance services with lower utilization
- Backpressure, backpressure, backpressure
- Implement load shedding
- Circuit breakers
Reducing variance

+ Separate services with different latency characteristics
+ Complex semantics → high performance variance; Use caution
+ You can’t do better than your backend. DB choices matter
+ No preemption (Node/Golang), no QoS
  + Cooperative yield
  + Break into small tasks
+ Be minded of GC, scheduled tasks, data structure shuffles, background tasks, etc.
Summary: what’s the cost of a millisecond?

- Much higher than you think, especially at the bottom of the stack
- High percentiles have disproportionate impact
  - Forget about averages
- Latency amplification is the most common reason for low utilization
- Need to actively combat amplification

Do yourself a favor and use a DB with a good percentile curve 😊😊
Thank you