Tail at Scale

With Personal Commentary

Yao Yue (@thinkingfish), **Twitter**

The Paper:

Tail at Scale

By Jeffrey Dean, Luiz André Barroso

[html, pdf, summary]

Models and Assumptions

1. A dependency tree of RPCs

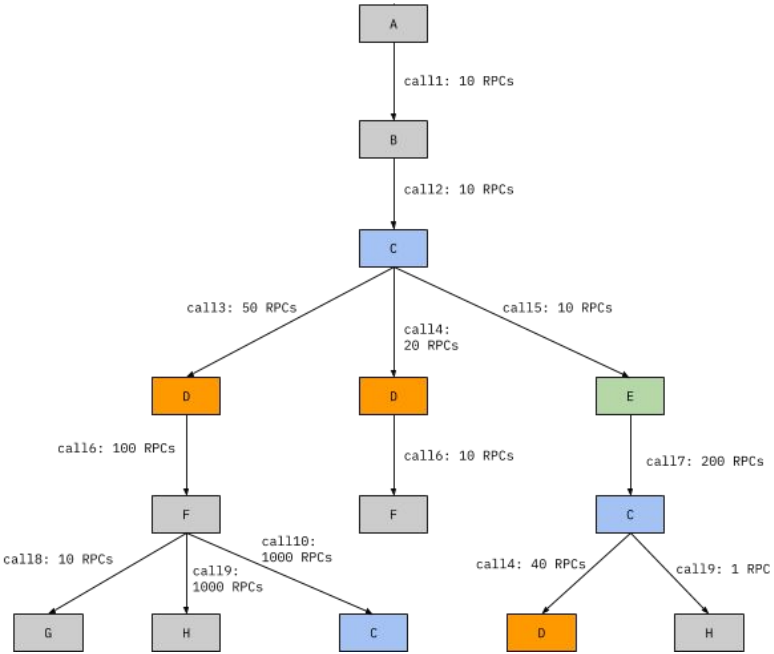
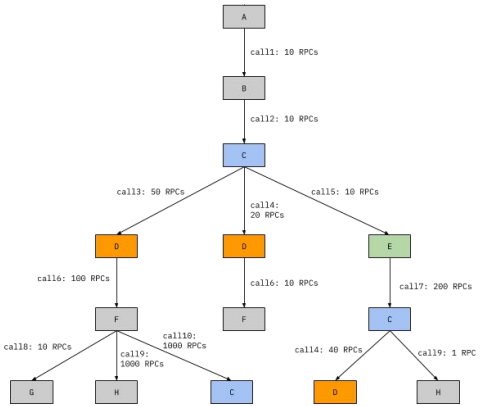


image by Dan Luu

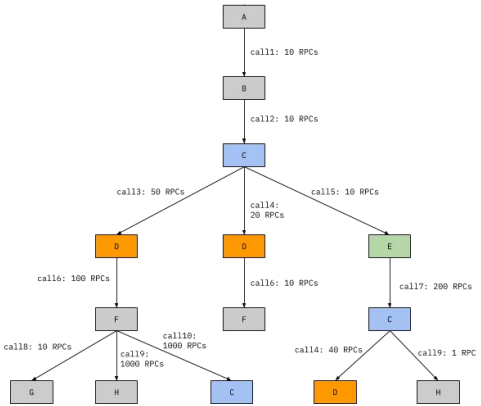
Models and Assumptions

1. A tree of RPCs
2. Impact of Tree Depth
    - Serial: additive impact



Models and Assumptions

1. A tree of RPCs
2. Impact of Tree Depth
3. Impact of Tree Width
    - Parallel (“fanout”): overall impact *???*

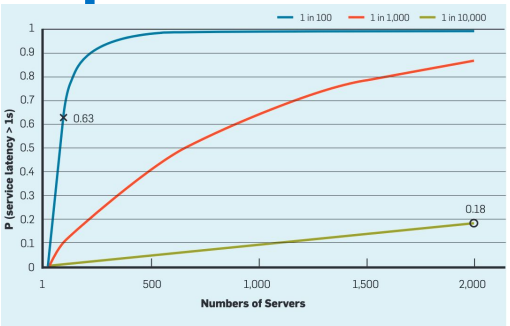


The Core Observation

1. Variability is a **certainty** (at scale)

2. Variability becomes **more pronounced** at scale

fanout	percentile
1	p99
10	p99**9**
100	p99**99**
1000	p99**999**



Overall latency distribution as a function of:
1. underlying latency distribution, and
2. fanout distribution.

How does variability manifest?

1. Resource starvation (“unfair wait”)
2. Queue depth (“fair wait”)
3. Degradation (“moving slowly”)

What drives variability?

1. Sharing
2. Bursty Behavior
    a. Background tasks, condition-triggered maintenance
    b. Inherent burstiness in demand / traffic
3. Tradeoff in physical environment

Comment and Disagreement (1)

Variability Sources

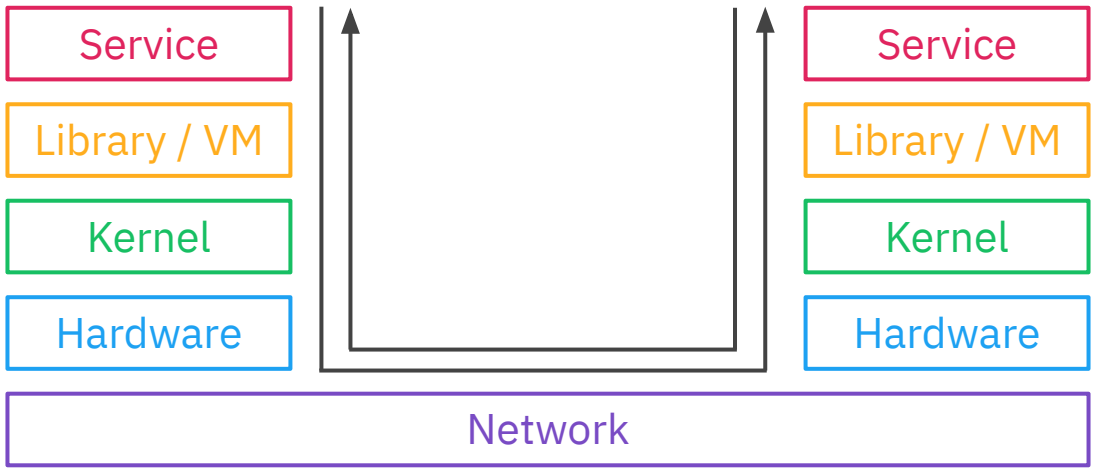
Shared Resources
Daemons
Global Resource Sharing
Maintenance Activities
Queueing
Power Limits
Garbage Collection
Energy Management

Comment

This list is not exhaustive, and it seems to be missing examples from high-level logic (such as GC in VMs, concurrent data structure, etc)

I attempted to classify things more generically in the previous slide.

RPC Stack


Service
Library / VM
Kernel
Hardware
Service
Library / VM
Kernel
Hardware
Network

Reduce variability: component builders

1. Service QoS
2. Synchronized Disruption
3. Normalizing work unit (Reduce head-of-line blocking)

Comment and Disagreement (2)

On synchronization

For large fan-out services, it is sometimes useful for the system to synchronize the background activity across many different machines. This synchronization enforces a brief burst of activity on each machine simultaneously, slow-ing only those interactive requests being handled during the brief period of background activity. In contrast, without synchronization, a few machines are always doing some background activity, pushing out the latency tail on all requests.

Comment

Don’t blindly try this at home. It could backfire.

The key here is using control of the timing as a way to contain exposure. That tends to work better if 1) a defer/avoidance mechanism is layered on top; 2) some redundancy is budgeted in.

Additional tricks for component builders

1. Minimize sharing
2. Limit bursts
3. Latency-optimized hardware configuration

Comment and Disagreement (3)

On caching

Missing in this discussion so far is any reference to caching. While effective caching layers can be useful, even a necessity in some systems, they do not directly address tail latency, aside from configurations where it is guaranteed that the entire working set of an application can reside in a cache.

Disagreement

Caching *is* an effective way of reducing fanout (to another system), and therefore, it directly addresses tail latency by reducing *how much* the tail latency of the fanouts impact overall latency.

For example, if cache hit rate is 90%, we effectively cut fanout (to the underlying storage) by a factor of 10. Since fanout is one of the two variables deciding overall latency distribution, caching has latency impact.

Mitigate variability: system builders

1. Hedged Requests
2. Tied Requests
3. Micro-partition, often *with* selective replication
4. Outlier detection (“latency-induced probation”)
5. Graceful degradation

Comment and Disagreement (4)

On Tied Requests

We call requests where servers perform cross-server status updates "tied requests."

Comment

Cross-server communication is a risky design pattern at scale.

This is more likely to succeed if requests are very expensive (relative to the exchange of small messages over network), are more likely near the top of the RPC tree.

Comment and Disagreement (5)
Hardware Trends

Exacerbating

- Machines getting bigger
- Deep memory hierarchy, NUMA
- Core network oversubscription

Alleviating

- Hardware offloading (of software functionalities)
- Resource isolation
- Disaggregated resources (e.g. network attached storage)

Bonus: Responding to Slowness

1. Timeout
2. Retry
3. The Snowball Effect

Detecting “grey failures” is one of the biggest challenges in distributed systems.

I have a whole talk about it...

Lies, Damned Lies, and Timeouts