Leveraging Queueing Theory and OS Profiling to Reduce Application Latency

Anshul Gandhi

Assistant Professor
Computer Science Dept.

Amoghavarsha Suresh

Ph.D. Student
Computer Science Dept.

PACE
Stony Brook University

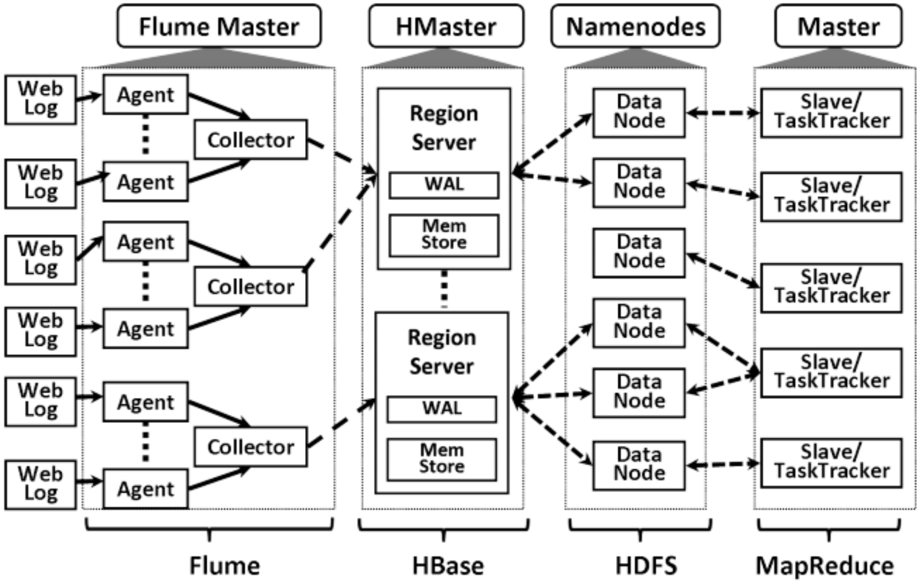
High-Level Motivation for this Tutorial

- Online (or web) applications are everywhere
- Such apps are interactive, responsive (sub-second latency)
- **Latency** is a critical metric

amazon Google YouTube

Applications are Complex

- Today’s online services consist of several components
- To optimize end-to-end latency, where should one start looking?



Goal: Achieving Low Latency

- Common approach: *underutilize* servers
- Other approaches: shorten the *critical path*
  - ➢ Chronos (SOCC'12): User-level networking, bypass kernel
  - ➢ UCR (ICPP'11): RDMA-capable Memcached
  - ➢ Tales of the Tail (SOCC'14): Real-time scheduling
  - ➢ Warehouse-scale computers (ISCA'15): Hardware specialization

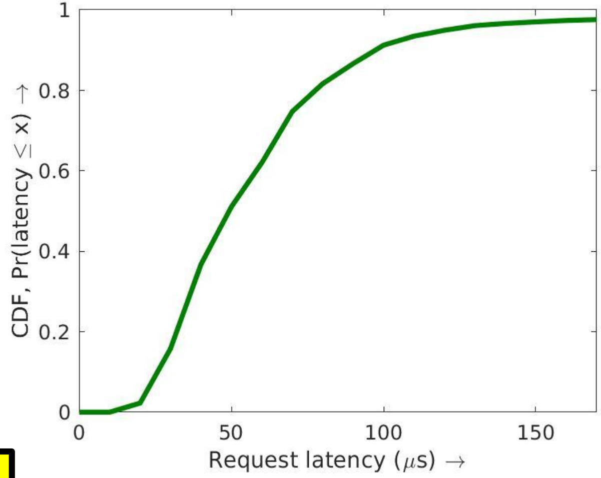
- All these approaches ignore a key issue: **variability**

Significance of Variability

- Request processing times are highly variable
- Harder to obtain low tail latencies

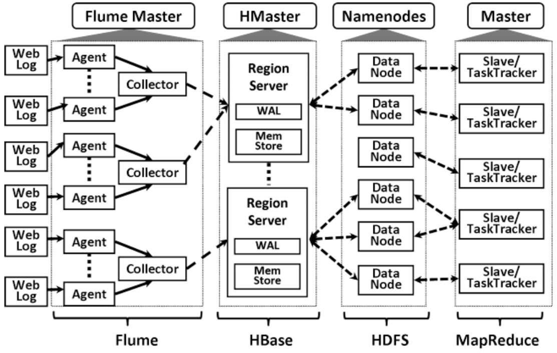
- But, variability represents an **opportunity**

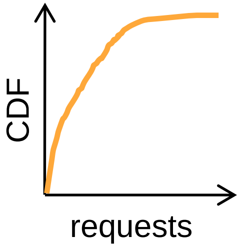
Our focus in this tutorial is on *directly* targeting a reduction in variability to improve latency

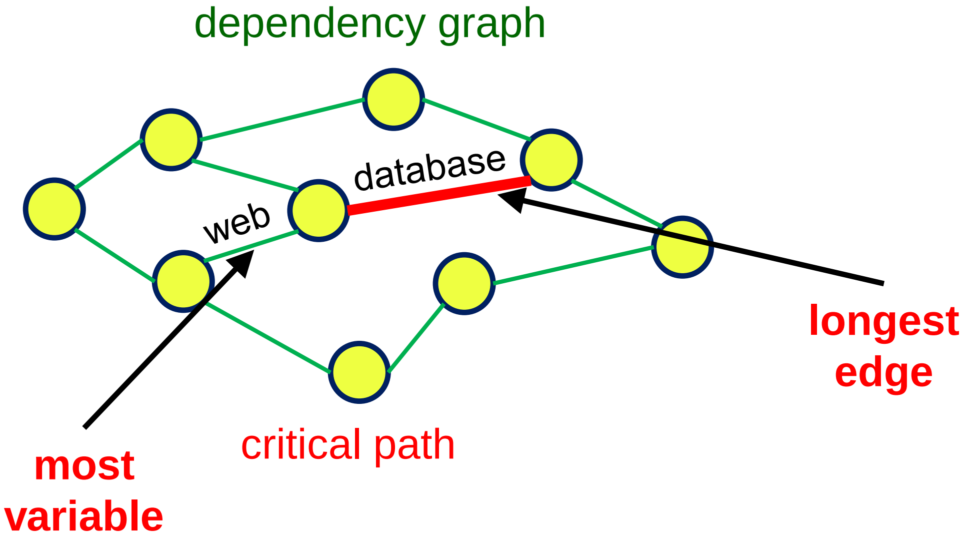


CDF of Memcached request latency

Significance of Variability

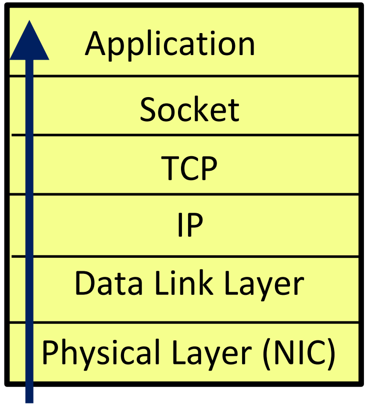




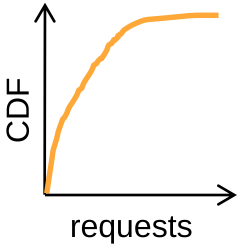


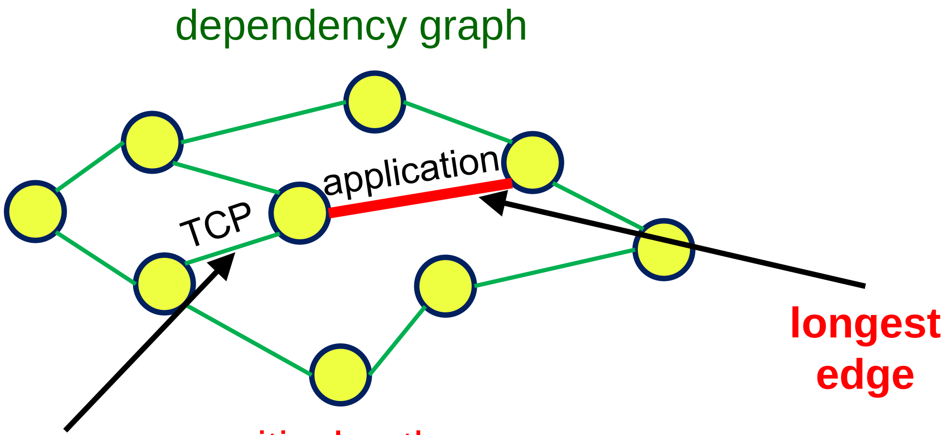
Variability represents an opportunity for reducing latency

Goal of this Tutorial



SERVER





most variable

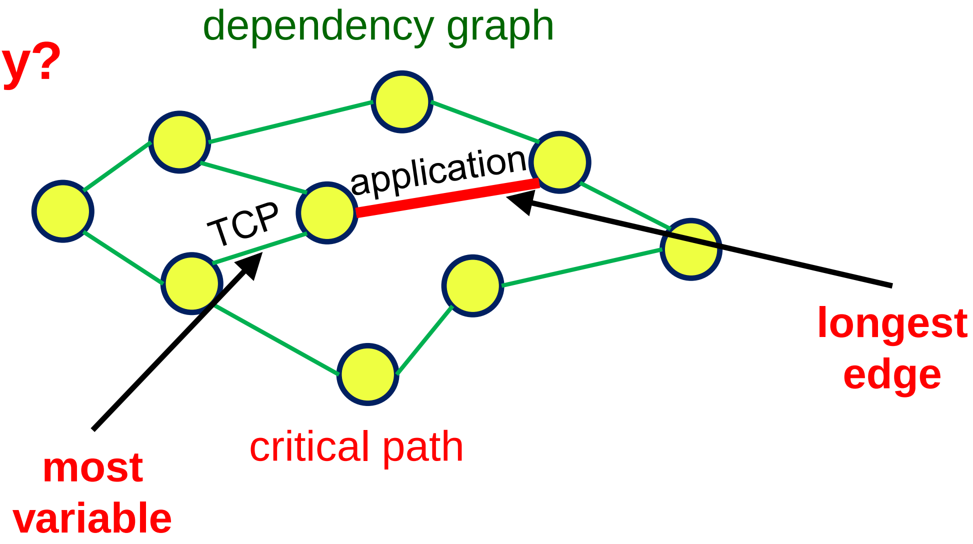
critical path

Reduce end-to-end server latency by targeting per-stage variability

High-Level Outline of Tutorial

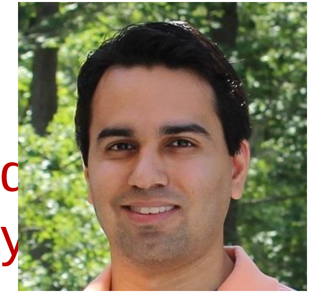
1. **How variability impacts latency?**
   - Why our approach works

2. **How to mitigate variability?**
   - How to apply our approach



Outline of Tutorial

Part 1: Queueing theory and practice



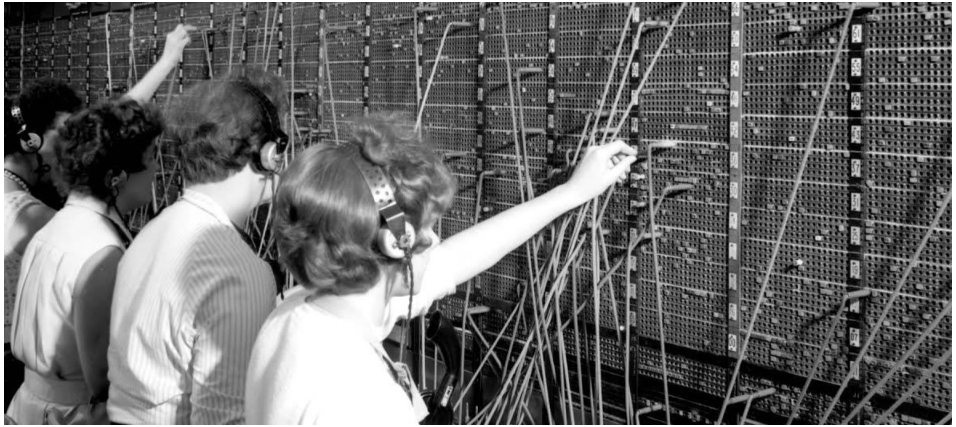
- Basics of queueing theory: arrivals, departures, queues
- Queueing models: M/M/1, M/M/k, M/G/1
- Useful lessons: latency vs. load, impact of variability, load
- Shortcomings: limiting assumptions, practical applicability
- Using queueing theory to detect application bottlenecks

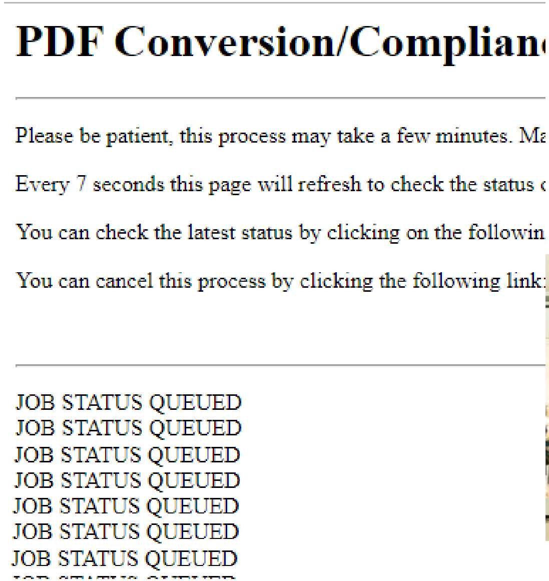
Part 2: Mitigating variability to reduce latency

- Application profiling: service time variability, stages of pro
- Control knobs: OS and application specific knobs to redu ty
- Case studies: Memcached, Apache web server; alternativ es
- Future work: multi-server, VMs, microservices

Queueing Theory Origins

- Early 1900s, by Erlang
- To analyze telephone exchanges
- Today, queues are everywhere!





PDF Conversion/Complian

Please be patient, this process may take a few minutes. Ma

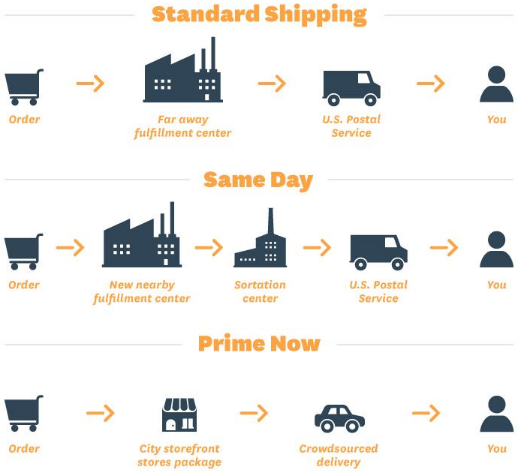
Every 7 seconds this page will refresh to check the status o

You can check the latest status by clicking on the followin

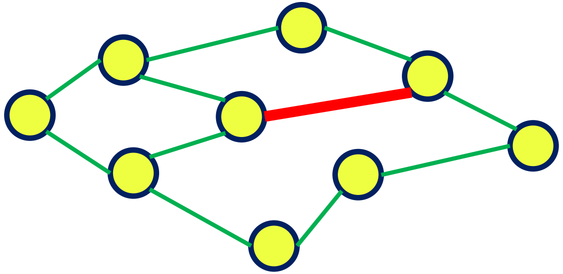
You can cancel this process by clicking the following link:

```
JOB STATUS QUEUED
JOB STATUS QUEUED
JOB STATUS QUEUED
JOB STATUS QUEUED
JOB STATUS QUEUED
JOB STATUS QUEUED
JOB STATUS QUEUED
```

Popular Applications of Queueing Theory



How Queueing Theory fits into this Tutorial

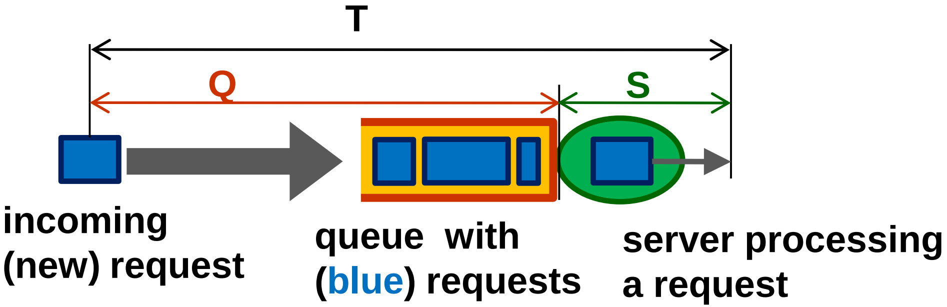


- Use queueing theory to analyze the impact of variability on latency
- Model each component as a queueing system
  - ➢ Example, packet processing at the NIC
  - ➢ Example, an entire server in a multi-tier deployment

Queueing Theory Basics

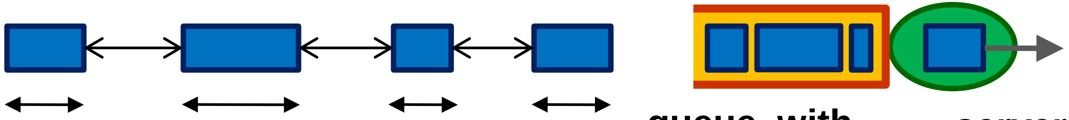
- Single-server, First-Come-First-Serve (FCFS)
- External arrivals, open-loop system

Request latency (**T**) = queueing time (**Q**) + service time (**S**)



How Queueing Theory Works

- Model latency (**T**) as a function of two processes or random variables:
  - Inter-arrival time, **IAT**, time between requests
    - 1/E[IAT] = λ requests/sec (average arrival rate)
  - Service time, **ST**, size of a request
    - 1/E[ST] = μ requests/sec (average service rate)
- Can also model number of requests in system (**N**) or queue (**N**$_Q$)

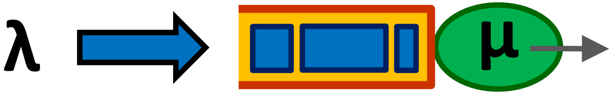


Arrivals and Services

- ➢ 1/E[IAT] = λ requests/sec (average arrival rate)
- ➢ 1/E[ST] = μ requests/sec (average service rate)

- ➢ Assume λ < μ always
- ➢ Why? What if λ > μ ??

- 4 GHz server
- Single-threaded CPU-intensive job requiring 1 Gigacycles to complete
- **E[ST] = ?? seconds**
- **μ = ? req/s**



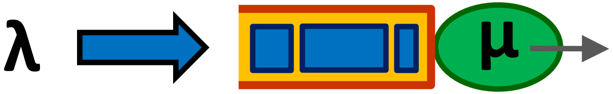
System Load

- ➢ 1/E[IAT] = λ requests/sec (average arrival rate)
- ➢ 1/E[ST] = μ requests/sec (average service rate)

Load (ρ) = E[ST]/E[IAT] = λ/μ

- E[ST] = 1/4 seconds (μ = 4 req/s)
- λ = 2 req/s
- **ρ = ??**

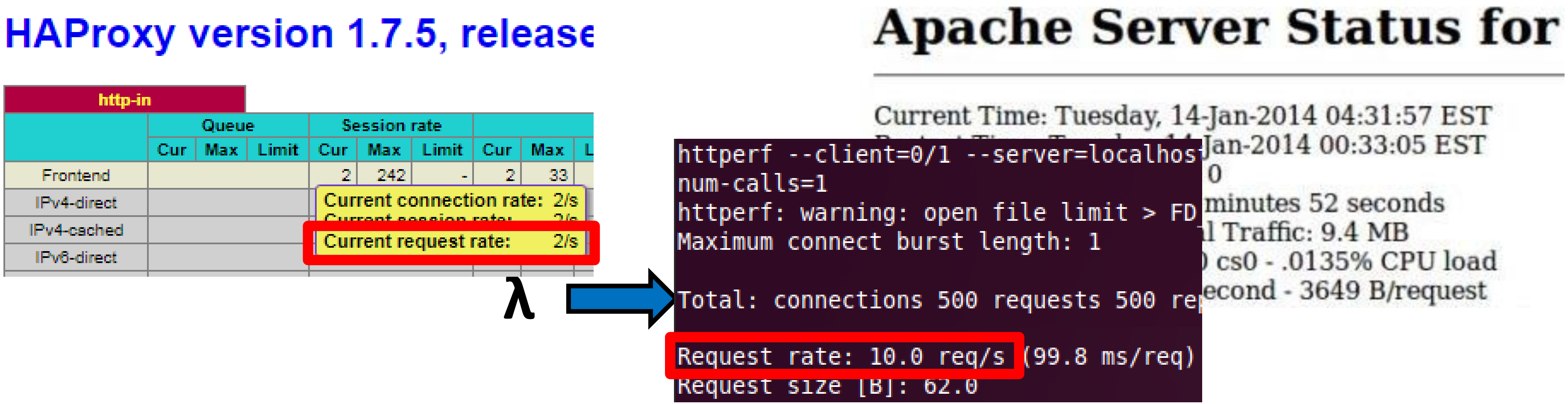
- ➢ Average incoming work/sec
- ➢ Note, ρ < 1



In Practice: Arrivals and Services

- λ and μ are key parameters of queueing models
- But how to obtain these in practice? Not always readily available.

1. λ is **average arrival rate:** measurable at load balancer or load generator



HAProxy version 1.7.5, release

http-in								
	Queue			Session rate				
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max
Frontend				2	242	-	2	33
IPv4-direct								
IPv4-cached								
IPv6-direct								

Current connection rate: 2/s
Current session rate: 2/s
Current request rate: 2/s

```
httperf --client=0/1 --server=localhos
num-calls=1
httperf: warning: open file limit > FD
Maximum connect burst length: 1

Total: connections 500 requests 500 re

Request rate: 10.0 req/s (99.8 ms/req)
Request size [B]: 62.0
```

Apache Server Status for

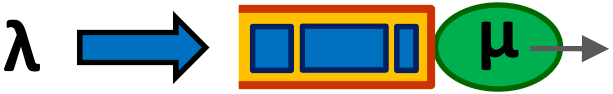
Current Time: Tuesday, 14-Jan-2014 04:31:57 EST
Jan-2014 00:33:05 EST
0
minutes 52 seconds
l Traffic: 9.4 MB
) cs0 - .0135% CPU load
econd - 3649 B/request

In Practice: Arrivals and Services

- λ and μ are key parameters of queueing models
- But how to obtain these in practice? Not always readily available.

2. μ is **average service rate**

μ is same as throughput??

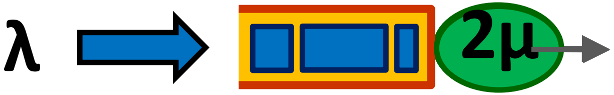


In Practice: What About Throughput?

- Throughput is **average rate at which requests are serviced**

- Avg. arrival rate λ req/s
- Avg. service rate μ req/s
- Assume no losses
- **Peak throughput = ??**
- **Throughput = ??**

- Avg. arrival rate λ req/s
- Avg. service rate 2μ req/s
- Assume no losses
- **Peak throughput = ??**
- **Throughput = ??**

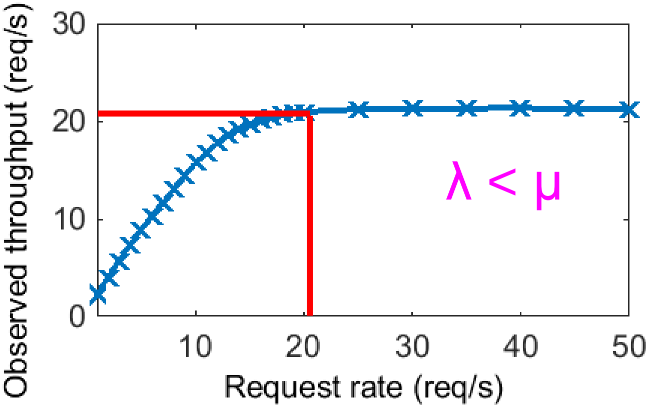


In Practice: Arrivals and Services

- λ and μ are key parameters of queueing models
- But how to obtain these in practice? Not always readily available.

2. μ is **average service rate**

μ is same as *peak throughput*



- μ = 1/E[ST]
- ST: time to service request (no queueing)
- **Measure E[ST] and set μ = 1/E[ST]**

Outline of Tutorial

Part 1: Queueing theory and practice

- Basics of queueing theory: arrivals, departures, queues
- Queueing models: M/M/1, M/M/k, M/G/1
- Useful lessons: latency vs. load, impact of variability, load balancing
- Shortcomings: limiting assumptions, practical applicability
- Using queueing theory to detect application bottlenecks

Queueing Models

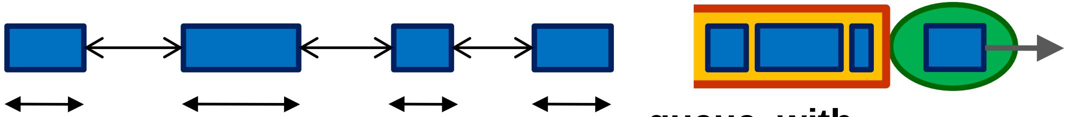
- Model latency (**T**) as a function of two processes or random variables:
  - ➢ Inter-arrival time, **IAT**, time between requests
  - ➢ Service time, **ST**, size of a request

- Queueing model: D_{IAT}/D_{ST}/1 model

distribution of IAT

single server

distribution of ST

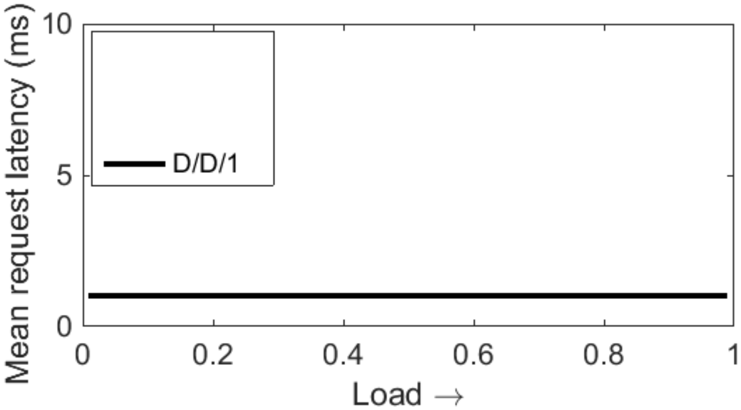


queue with (blue) requests

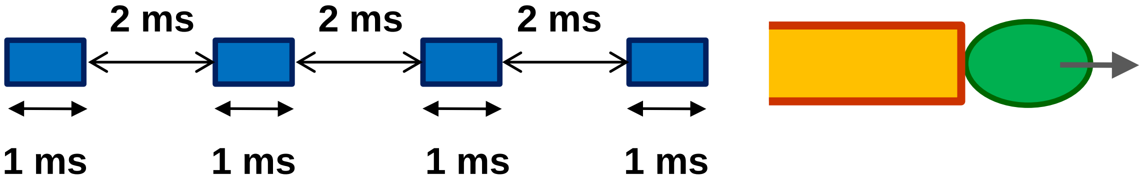
server processing a request

Significance of the IAT and ST Distribution

- Common distributions:
  - ➢ D: Deterministic (zero var)

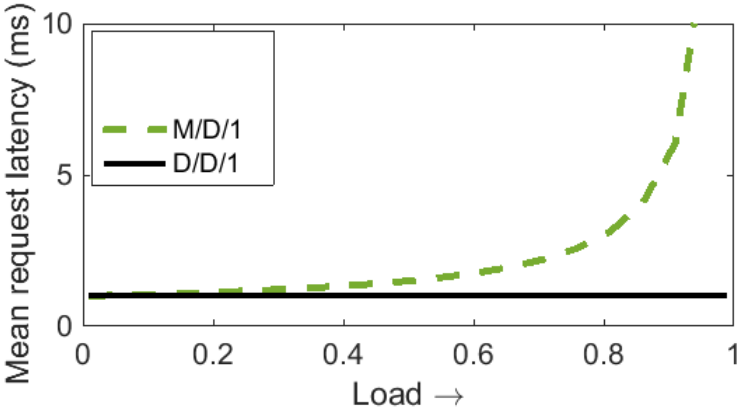


E[ST] = 1 ms;   E[IAT] = Load/E[ST]



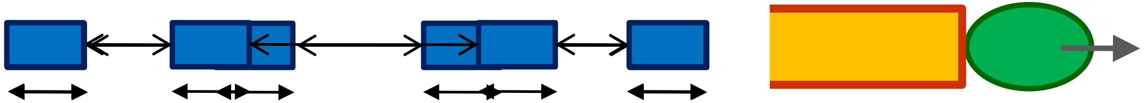
Significance of the IAT and ST Distribution

- Common distributions:
  - D: Deterministic (zero var)
  - M: Exponential (medium var)



E[ST] = 1 ms; E[IAT] = Load/E[ST]

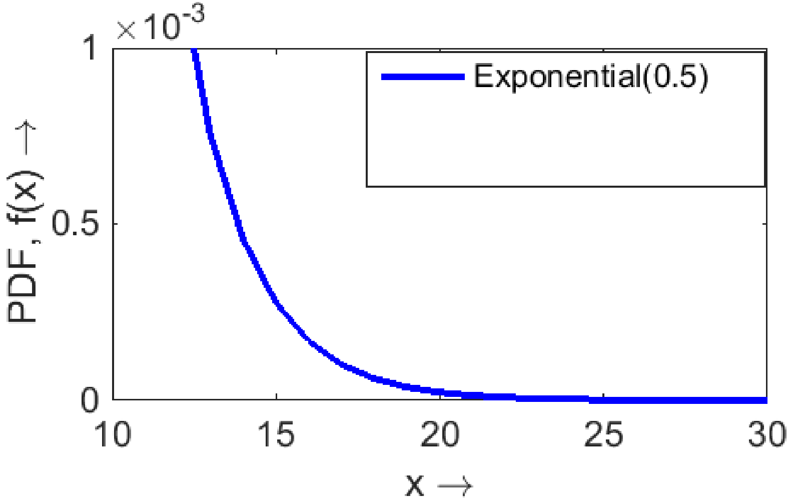
M/D/1 model



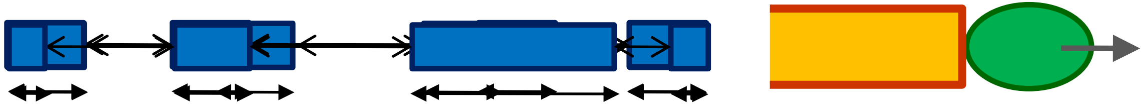
IAT and ST Distributions

- Common distributions:
  - D: Deterministic (zero var)
  - M: Exponential (medium var)

$$f(x) \,\square\, \frac{1}{e^{x}}$$



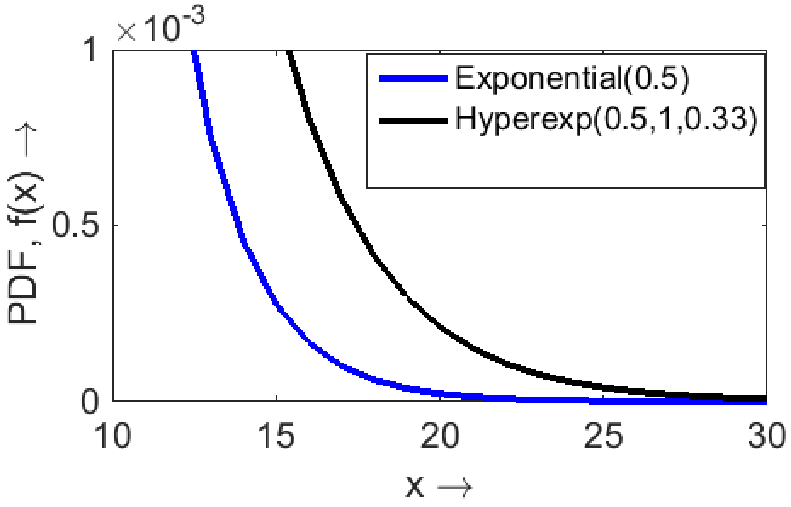
Mean = 2



IAT and ST Distributions

- Common distributions:
  - D: Deterministic (zero var)
  - M: Exponential (medium var)
  - H2: Hyper-exponential (tunable)

$$H_2 = \begin{cases} Exp(\lambda_1) \text{ w.p. p} \\ Exp(\lambda_2) \text{ w.p. (1-p)} \end{cases}$$

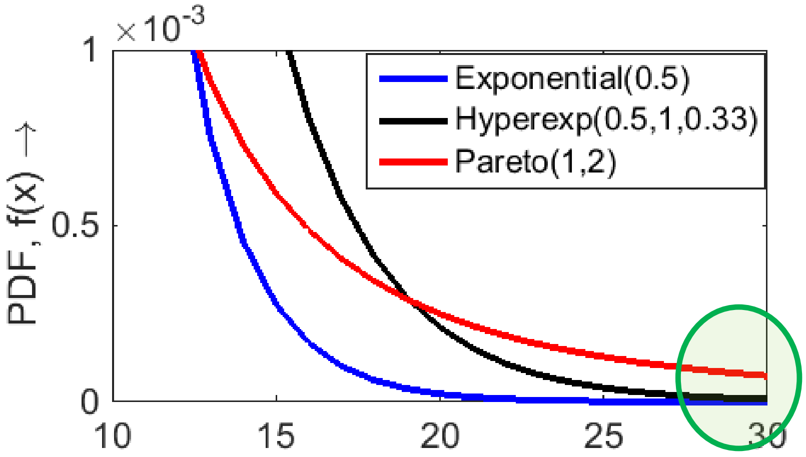


Mean = 2

IAT and ST Distributions

- Common distributions:
  - ➢ D: Deterministic (zero var)
  - ➢ M: Exponential (medium var)
  - ➢ H2: Hyper-exponential (tunable)
  - ➢ Pareto (high var)

$$f(x) \square \frac{1}{x^{\alpha+1}}$$



Mean = 2

“Heavy” tail

Heavy tail distribution has a tail that is heavier than that of an exponential

Queueing Models: Results

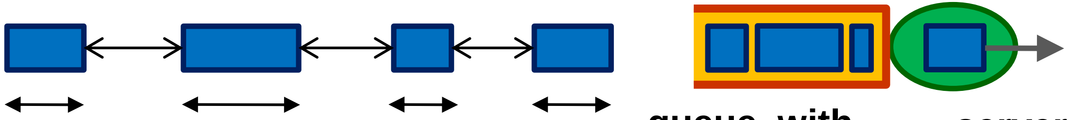
- Model latency (**T**) as a function of two processes or random variables:
  - ➢ Inter-arrival time, **IAT**, time between requests
  - ➢ Service time, **ST**, size of a request

- Queueing model: $D_{IAT}/D_{ST}/1$ model

distribution of IAT

single server

distribution of ST

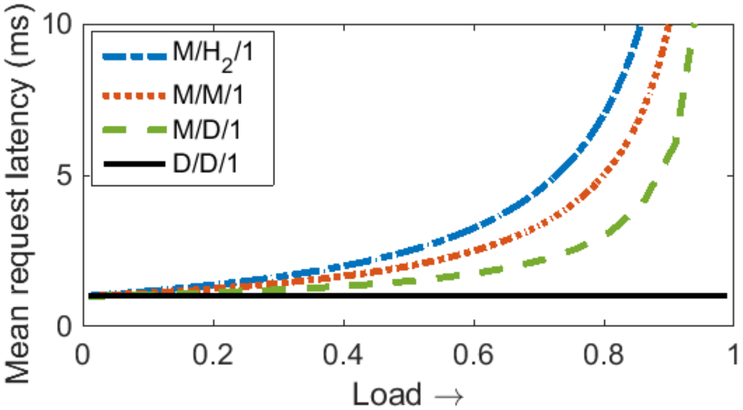


queue with (blue) requests

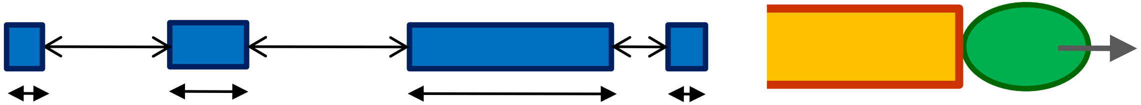
server processing a request

Queueing Models: Results

- Common distributions:
  - D: Deterministic (zero var)
  - M: Exponential (medium var)
  - H_2: Hyper-exponential (high var)
  - Pareto (high var)
  - **G: General distribution**



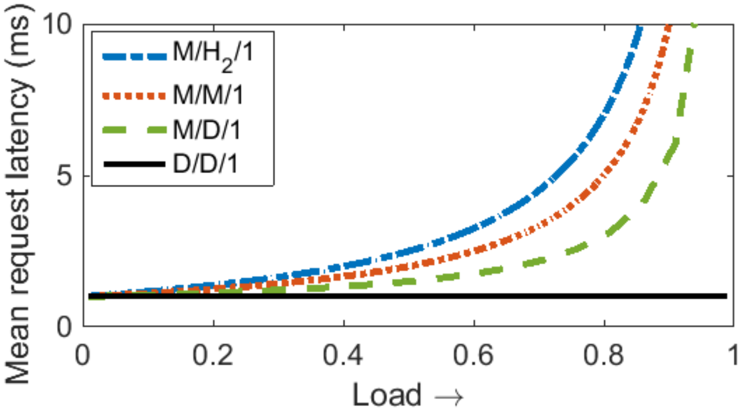
E[ST] = 1 ms;   E[IAT] = Load/E[ST]



A. Suresh and A. Gandhi, Using Variability as a Guiding Principle to Reduce Latency in Web Applications via OS Profiling, WWW 2019

Queueing Models: Results

- Latency rises non-linearly with load
- M/M/1: $E[T] = 1/(\mu - \lambda) = E[ST]/(1 - \boldsymbol{\rho})$
- $T_{95} = E[ST]*\ln(20)/(1 - \boldsymbol{\rho})$
- $T_x = E[ST]*\ln(1-.01x)/(1 - \boldsymbol{\rho})$



E[ST] = 1 ms;   E[IAT] = Load/E[ST]

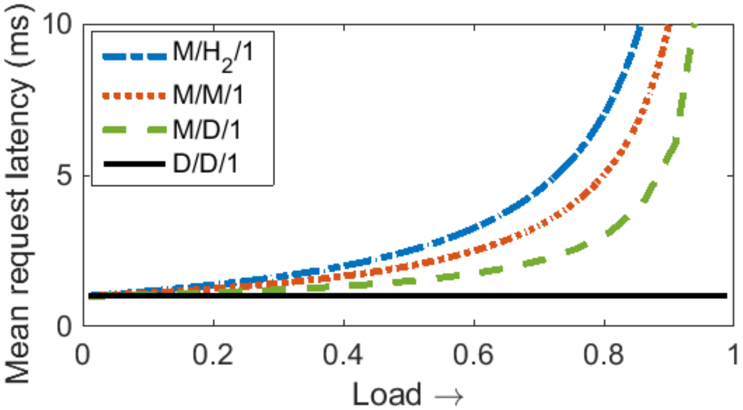
Takeaway 1

Latency ~ 1 / (1 - ρ)

Queueing Models: Results

- For a given load, latency increases with IAT and ST variability

- For a given load:
  $T_{M/H_2/1} > T_{M/M/1} > T_{M/D/1} > T_{D/D/1}$

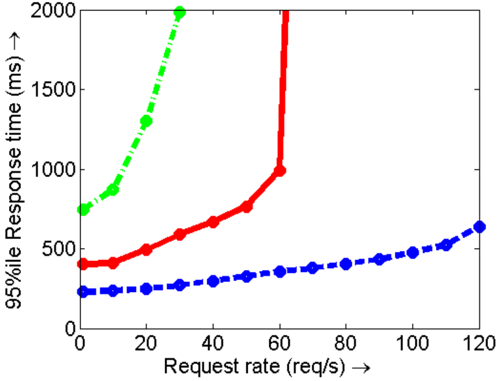


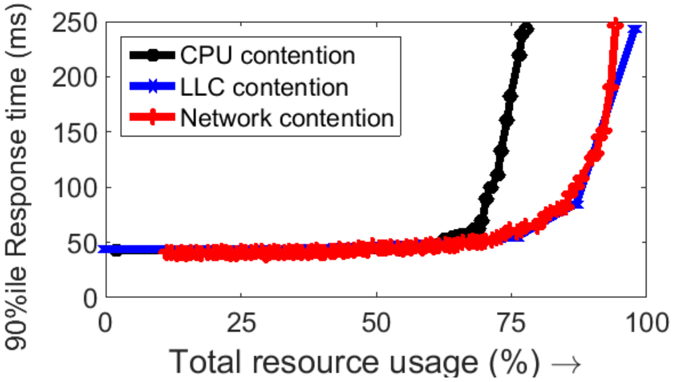
E[ST] = 1 ms;   E[IAT] = Load/E[ST]

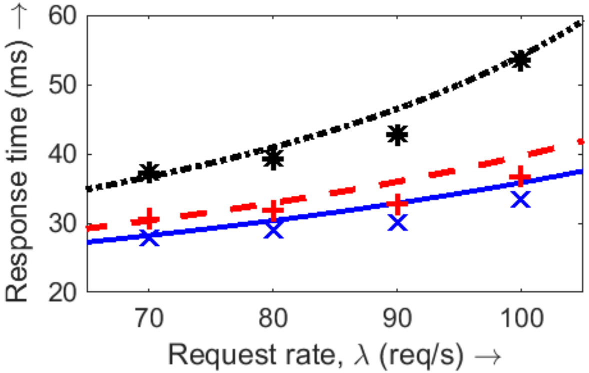
<u>Takeaway 2</u>

Latency increases with load *and* IAT and ST variability

In Practice: Queueing Models



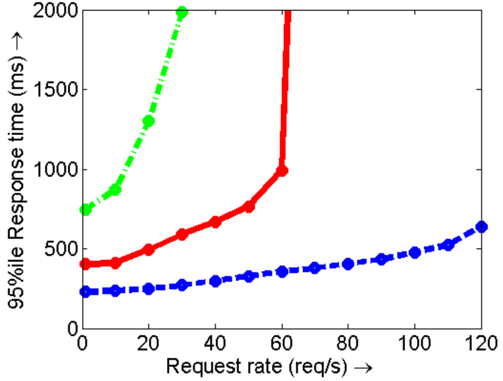


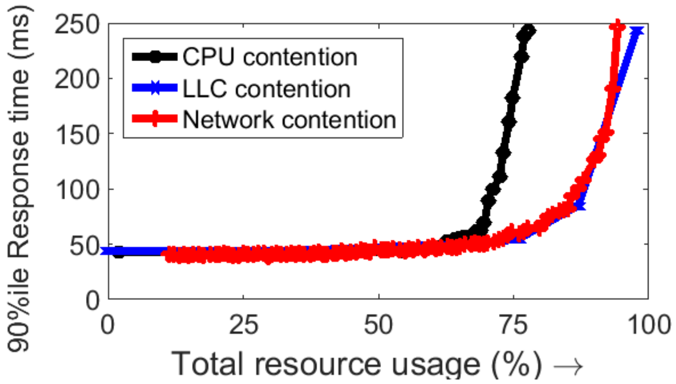


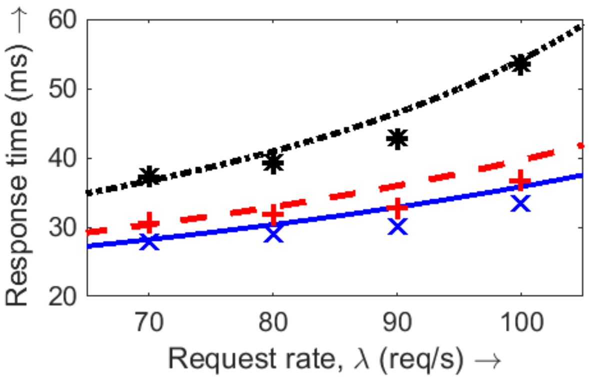
- In practice, latency ~ 1/(1 - ρ), and not latency ~ ρ
- However, in practice, **latency ≠ E[ST]/(1 - ρ)**
  - ➢ IAT and ST not always exponential
  - ➢ Network delays, resource contention

A. Gandhi et al., AutoScale, ACM Trans. Comp. Sys., 2012; S. Javadi et al., DIAL, ICAC 2017; S. Votke et al., Modeling and Analysis of Performance under Interference in the Cloud, Mascots 2017

In Practice: Queueing Models







- A better approximation in practice:

$$T = \alpha_1 + 1 \Big/ \left(1 - \alpha_2 \cdot \rho\right)^{\alpha_3}$$

network delays

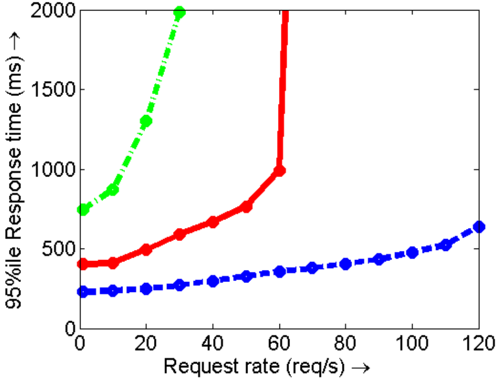
resource contention

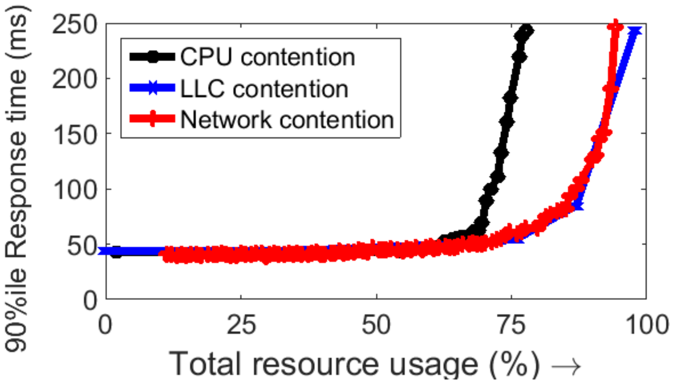
heavy-tail distributions

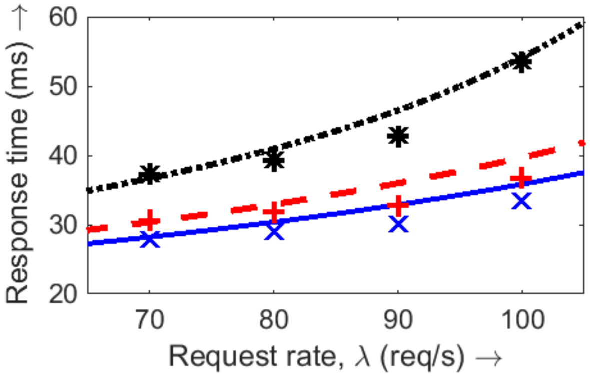
Solve for α via regression or control theory

S. Javadi et al., DIAL, ICAC 2017; A. Gandhi et al., Providing Performance Guarantees for Cloud-deployed Applications, IEEE Trans. Cloud Computing, 2018

In Practice: Queueing Models







Takeaway 3

$$T = \alpha_1 + \frac{1}{\left(1 - \alpha_2 \cdot \rho\right)^{\alpha_3}}$$

Queueing models are *not* meant to be used out-of-the-box

In Practice: IAT and ST distributions

- Common distributions:
  - D: Deterministic (zero var)
  - M: Exponential (medium var)
  - H2: Hyper-exponential (tunable)
  - Pareto (high var)

Which distribution does my IAT and ST follow?

Distribution fitting to derive the best fit for your data!

$H_2/H_2/1$ model

Takeaway 4

The H2 distribution can be tuned via its parameters to provide an adequate fit for IAT and ST

M. Wajahat et al., Distribution Fitting and Performance Modeling for Storage Traces, Mascots 2019 (Best Paper)

Multi-Server Queueing Models

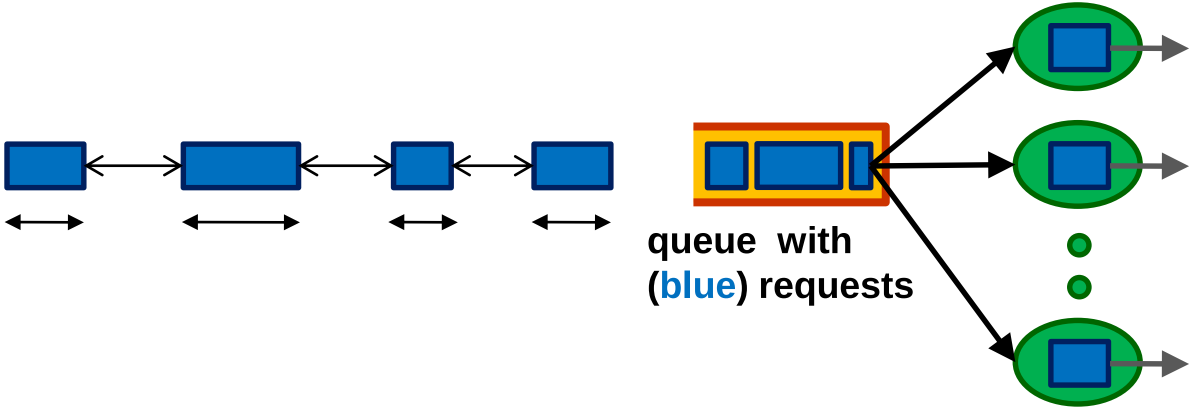
- Today’s applications employ a **cluster** of servers to serve the workload

- Queueing model: $D_{IAT}/D_{ST}/k$ model

distribution of IAT

k servers

distribution of ST



queue with (blue) requests

Scheduling: idle server picks request from head-of-queue

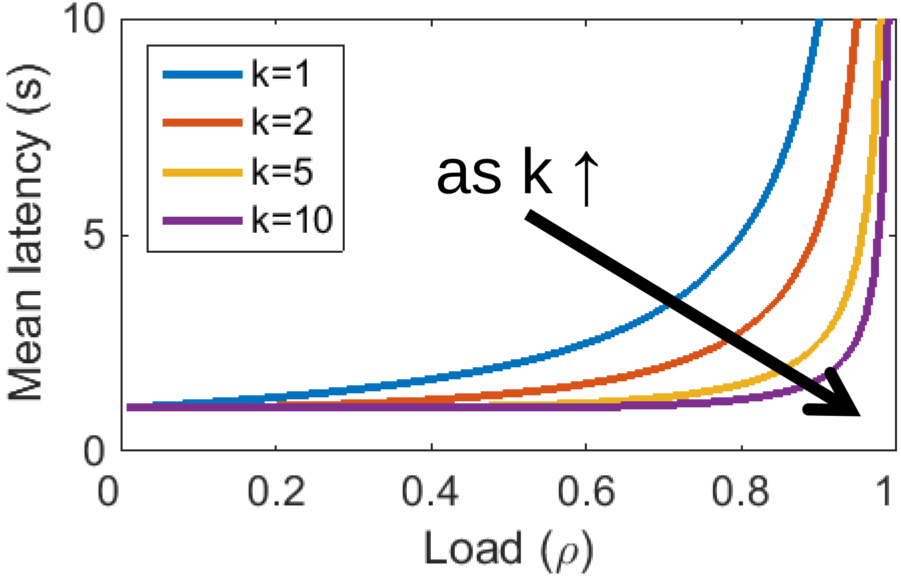
cluster of servers processing requests

Multi-Server Queueing Models: Results

- M/M/k

Takeaway 5

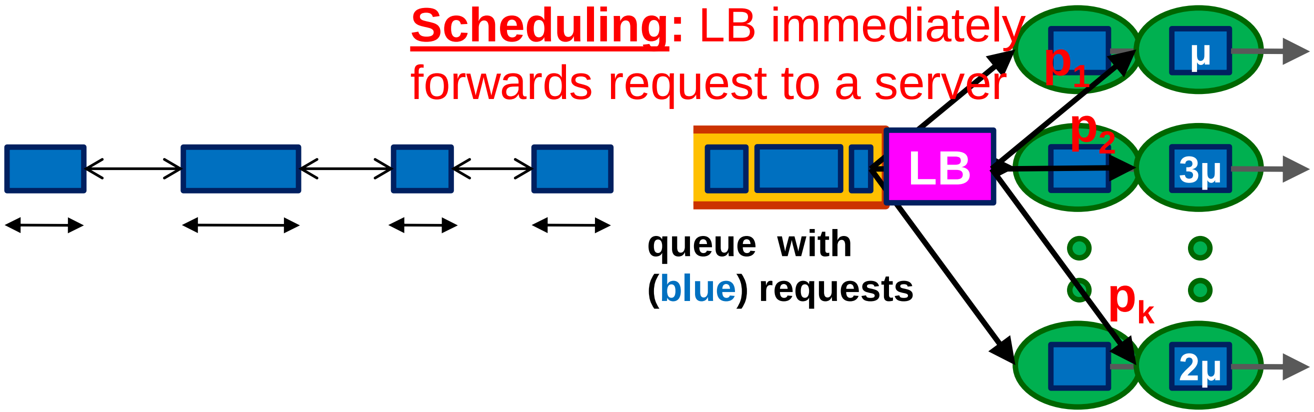
- **Pr(all k servers busy) ~ ρ^k**
- **With more servers, we can better handle load variations**



$\rho = \lambda/k\mu < 1$

In Practice: Multi-Server Queueing Models

- How to load balance among heterogeneous, processor sharing, servers?
  - *Proportional to their service rates??*
  - *No!*

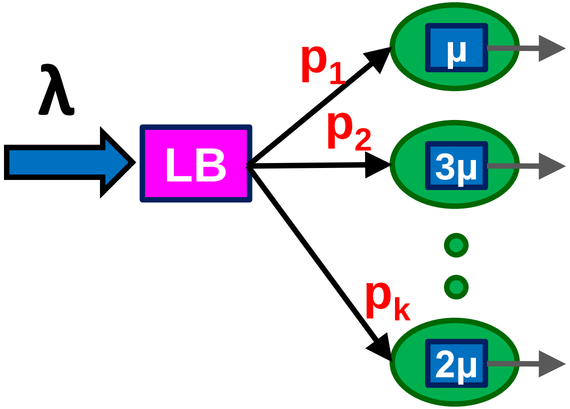

Scheduling: LB immediately forwards request to a server
LB
queue with (blue) requests
p_1
p_2
p_k
μ
3μ
2μ

In Practice: Multi-Server Queueing Models

- How to load balance among heterogeneous, processor sharing, servers?
  - ➢ Send *more-than-proportional* load to faster servers
  - ➢ Send *less-than-proportional* load to slower servers

Takeaway 6

$$p_i^* = \frac{\mu_i \cdot \sum_j \sqrt{\mu_j} - \sqrt{\mu_i} \cdot \sum_j \mu_j + \lambda \cdot \sqrt{\mu_i}}{\left(\lambda \cdot \sum_j \sqrt{\mu_j}\right)}$$



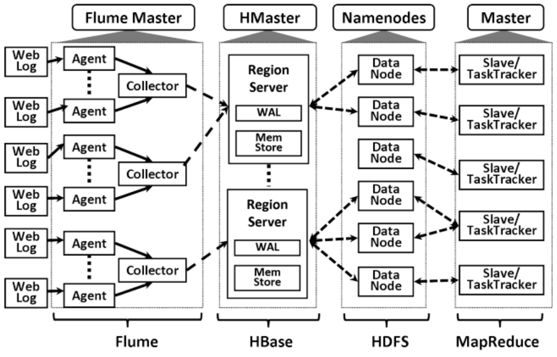
S. Javadi et al., DIAL, ICAC 2017
A. Gandhi et al., HALO, Mascots 2015

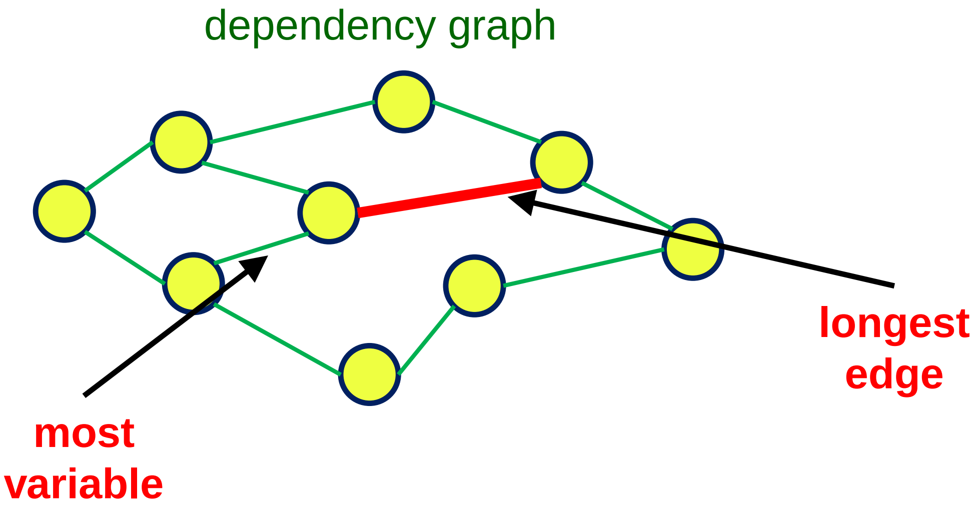
Outline of Tutorial

Part 1: Queueing theory and practice

- Basics of queueing theory: arrivals, departures, queues
- Queueing models: M/M/1, M/M/k, M/G/1
- Useful lessons: latency vs. load, impact of variability, load balancing
- Shortcomings: limiting assumptions, practical applicability
- Using queueing theory to detect application bottlenecks

Back to Variability

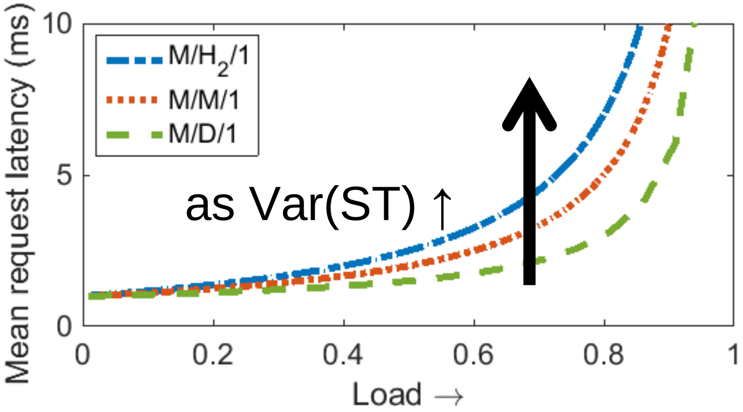




- ~~Inter-arrival time, **IAT**, time between requests~~
- Service time, **ST**, size of a request

Service Time Variability

- D: Deterministic (zero var)
- M: Exponential (medium var)
- H_2: Hyper-exponential (high var)



E[ST] = 1 ms;   E[IAT] = Load/E[ST]

- Service time, **ST**, size of a request

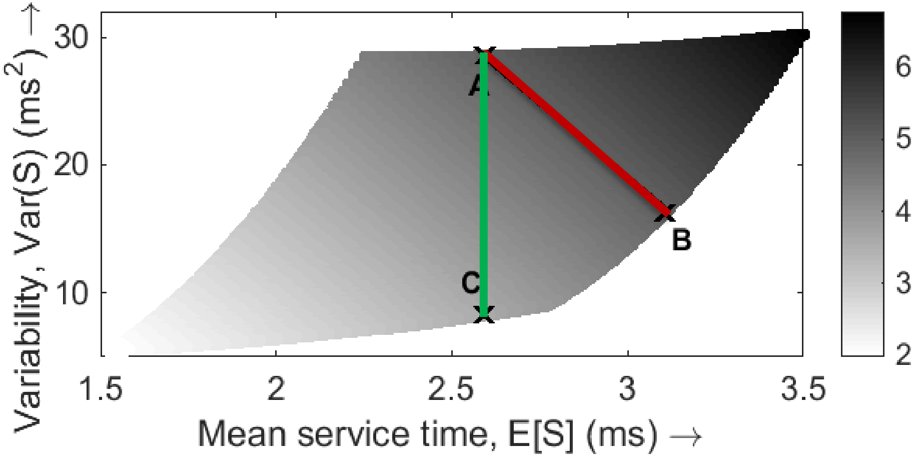
- **Var(ST)** is important
- But what about **E[ST]** ?

Impact of Var(ST) and E[ST] on Latency

M/G/1 model (P-K formula)

$$\mathbf{E[T]} = \frac{\mathbf{Var(ST)}}{2\cdot \mathrm{E[IAT]}\cdot(1-\rho)} + \frac{\mathbf{E[ST]}\cdot(2-\rho)}{2(1-\rho)}$$

- T: Latency
- ST: Service time – size of a request
- IAT: Inter-arrival time
- ρ: load (work/sec)



Latency heatmap as function of Var(ST), E[ST]

Takeaway 7

Reducing Var(ST), even at the expense of E[ST], can significantly reduce latency

Outline of Tutorial

Takeaway 1

Latency ~ 1 / (1 - ρ)

eueing e

Takeaway 2

Latency increases with load *and* IAT and ST variability

Takeaway 3

$$T = \alpha_1 + 1 \Big/ (1 - \alpha_2 \cdot \rho)^{\alpha_3}$$

- Basics of queueing res, que
- Queueing models: M/M/1, M/M/k, M/G/1

o a g a r ot

Takeaway 4

The H2 distribution can be tuned via its parameters to provide an adequate fit for IAT and ST

Takeaway 5

- **Pr(all k servers busy) ~ ρ^k**
- **With more servers, we can better handle load variations**

Takeaway 6

$$p_i^* = \frac{\mu_i \cdot \sum_j \sqrt{\mu_j} - \sqrt{\mu_i} \cdot \sum_j \mu_j + \lambda \cdot \sqrt{\mu_i}}{\left(\lambda \cdot \sum_j \sqrt{\mu_j}\right)}$$

Part 2:

Takeaway 7

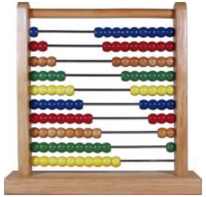
Reducing Var(ST), even at the expense of E[ST], can significantly reduce latency

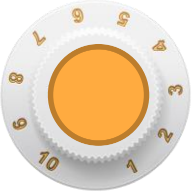
- Applica
- Control ty
- Case studies: Memcached, Apache web server; alternativ es
- Future work: multi-server, VMs, microservices

Solution Overview for Client-Server Web Systems

Step 1: Fine-grained probing to track request processing stages

Step 2: Compute variability at each stage to find bottlenecks



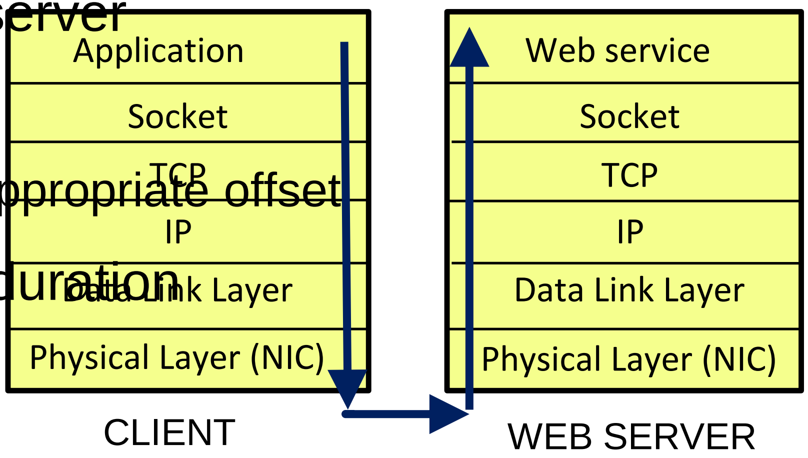


Step 3: Find appropriate control knobs to reduce variability

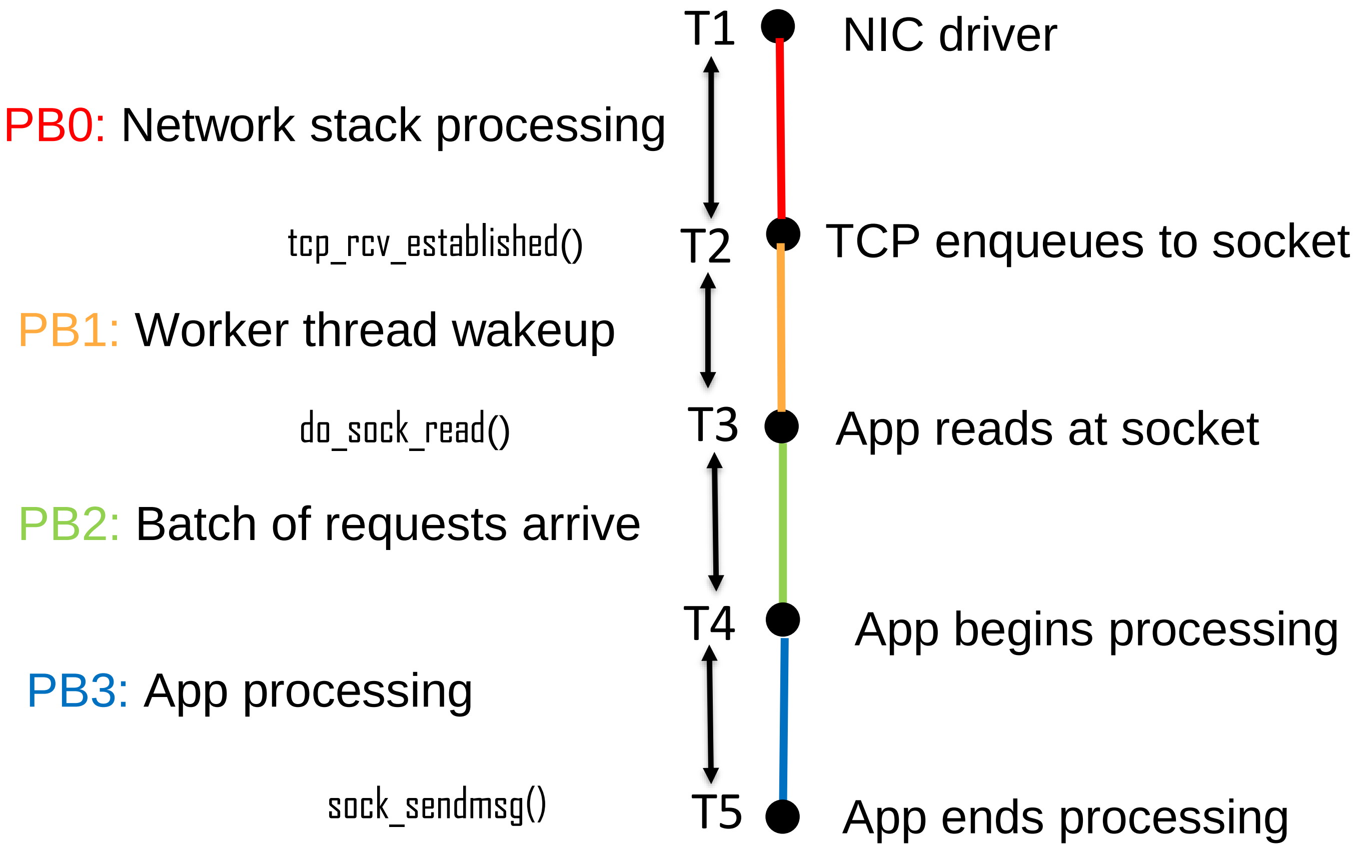
Objective: Use *Variability of Service Time* as a Guiding Principle to Reduce Application Latency

Fine-Grained Request Probing

- Timestamp the request as it traverses server
  - ➢ Append 64 bytes buffer to request
  - ➢ At stage boundaries, add timestamp at appropriate offset
- Use timestamps to compute per-stage duration



Fine-Grained Request Probing



T1 NIC driver

PB0: Network stack processing

tcp_rcv_established() T2 TCP enqueues to socket

PB1: Worker thread wakeup

do_sock_read() T3 App reads at socket

PB2: Batch of requests arrive

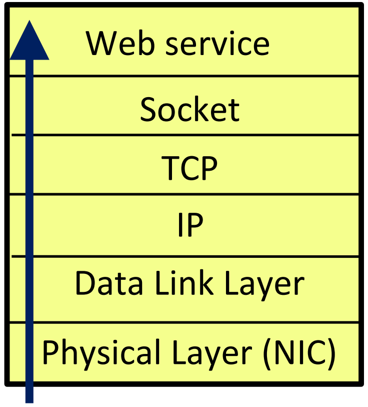
T4 App begins processing

PB3: App processing

sock_sendmsg() T5 App ends processing

Computing Variability of Service Time at Each Stage

- $\mathrm{Var}(S) = E[S^2] – (E[S])^2$
  - ➢ $E[S] \approx (s_1 + s_2 + \ldots + s_n)/n$; $E[S^2] \approx (s_1^2 + s_2^2 + \ldots + s_n^2)/n$
    - ▪ n requests
    - ▪ s_i: duration for request i
  - ➢ Only need running sum of duration (S) and its square (S^2)
  - ➢ Low overhead

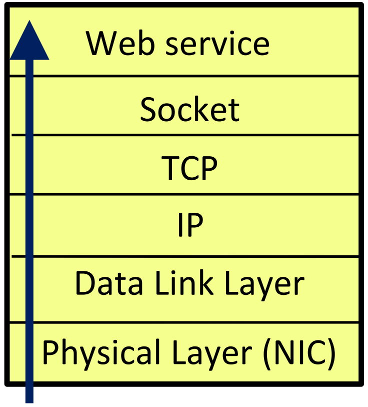


WEB SERVER

PB0: **Network stack processing**

Computing Variability of Service Time at Each Stage

- Running sum will result in large sums, especially E[S^2]
- Alternatively can use Welford’s online algorithm
- Need to record requests over a window *W*
- For a new sample x_{w+1} :
- Delta in means: $(\sum_{i=2}^{W+1} x_i - \sum_{i=1}^{W} x_i \ )/N$
- Delta in variance: $(x_{w}{}_{+}\,1{}_{-}\,x_{1})(xw - \mu_{w}{}_{+}\,1{}_{+}\,x_{1} - \mu_{w})$



WEB SERVER

PB0: **Network stack processing**

Finding A Control Knob

- Find service time (ST) variability of all the stages
- In the decreasing (highest first) order of ST variability, examine the functionality
- Reason what about the **functionality** and **implementation** makes it ***variable***
- **Control-Knob:** Change the implementation to reduce variability, while retaining functionality, for example
  - Introduce batching of constant size, to make service time predictable
  - Reducing interference from background threads by changing thread scheduling

Outline

Part 2: Mitigating variability to reduce latency

- Application profiling: service time variability, stages of processing
- Control knobs: OS and application specific knobs to reduce variability
- Case studies: Memcached, Apache web server; alternative strategies
- Future work: multi-server, VMs, microservices
- Conclusion

Methodology

Experimental setup:

- Server and Client: Intel Xeon 2620, 64GB DRAM,1Gbps via ToR switch
- Linux kernel version 3.16.7

Methodology:

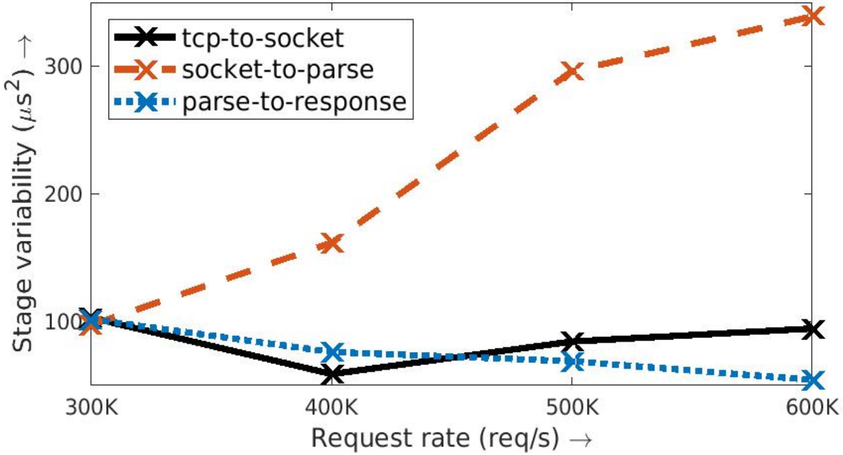
- Running sum of service time for each stage across all (10M) requests
- Averaged over 5 iterations

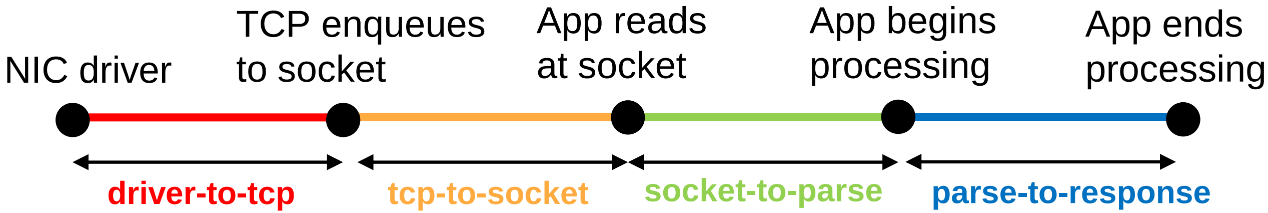
Applications:

- Memcached: In-memory, key-value store, event driven, multi-threaded
- Apache web server: Highly scalable, multi-process + multi-threaded

Memcached: High Throughput Configuration

- 5 worker threads on 5 cores
- 1 core used by LRU thread
- Bottleneck: **socket-to-parse**

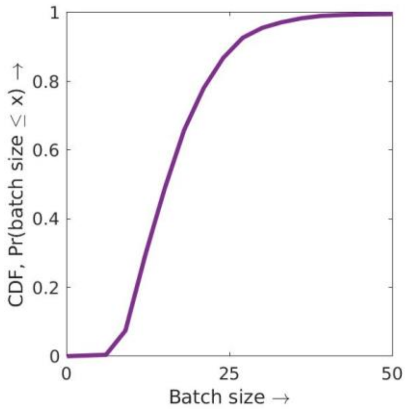




Bottleneck Analysis

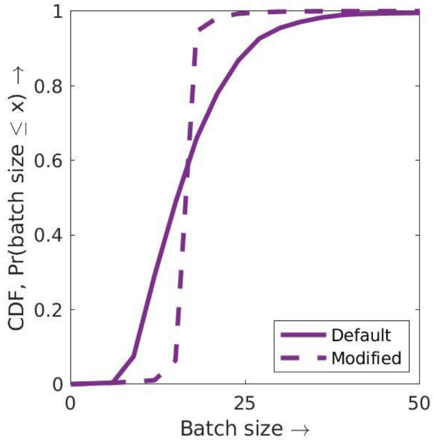
- Socket-to-parse: parsing the drained batch of requests from the socket, one request at a time (last request in batch has to wait a long time)
- Time taken in this stage is proportional to the size of the request batch

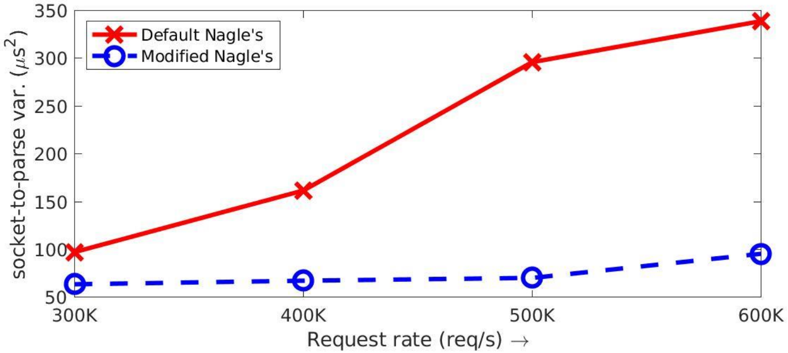
- Control knob: Nagle's algorithm at Client
  - ➢ Batch size determined by network conditions
  - ➢ Variable n/w conditions → batch size variability



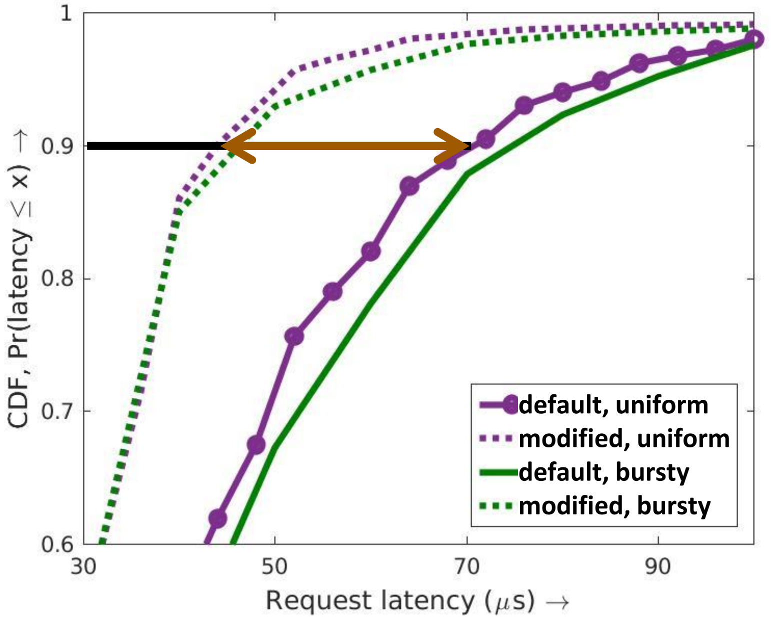
Finding the Control Knob

- Knob: admission control threshold (max wait time before batch is sent)
  - ➢ Threshold too small → too many small packets
  - ➢ Threshold too large → large delays
  - ➢ Determined empirically
- Significantly reduces batch size and stage variability





Improvement in Application Latency



Constant load (300K req/s)

- Mean latency improves by **24−26%**
- Tail latency improves by **34−40%**

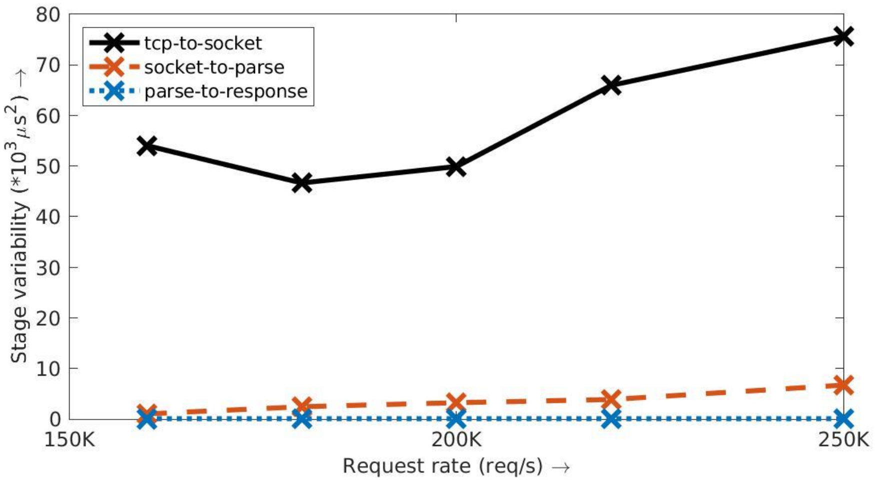
Facebook's VAR, APP, ETC traces

- Mean latency improvement: **14−20%**
- Tail latency improvement: **26−39%**

Lowering the variability does indeed help to reduce latency

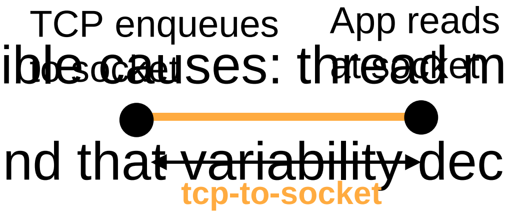
Memcached: Low Throughput Configuration

- 2 worker threads on 2 cores
- 1 core used by LRU thread
- Bottleneck: **tcp-to-socket**



Bottleneck analysis:

- **Tcp-to-socket**: end of TCP proc to app picking up request from socket
- Possible causes: thread migration, background processes
- We find that variability decreases as # cores (and load) increases



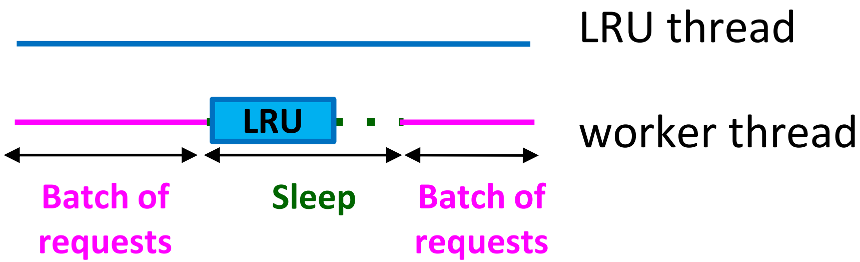
TCP enqueues to socket

App reads at socket

tcp-to-socket

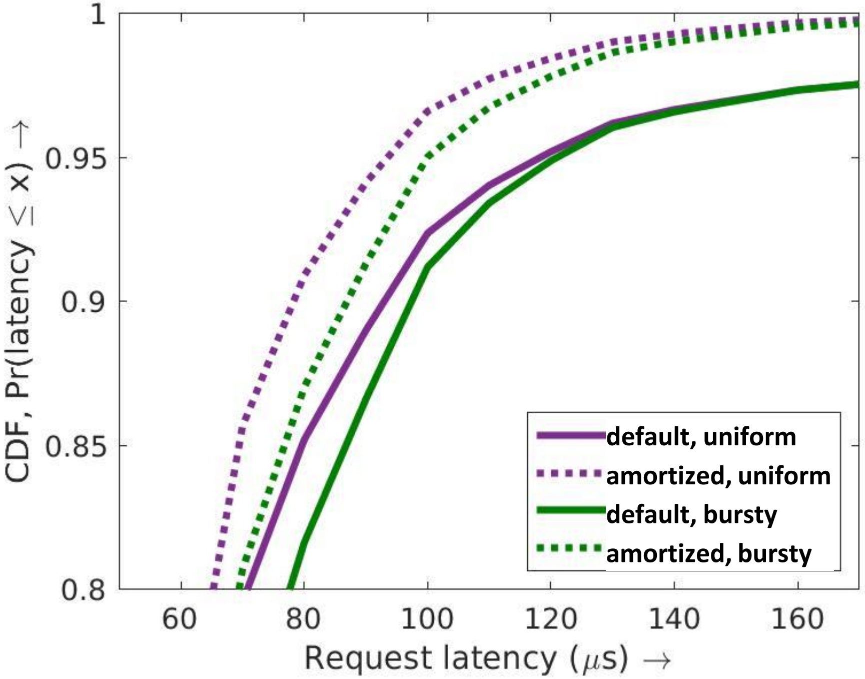
Finding the Control Knob

- Memcached LRU maintenance thread causes interference and variability
- **Control knob**: Move LRU maintenance to worker thread
- LRU maintenance should:
  - ➢ *Emulate default LRU work*
  - ➢ *Avoid stepping on future requests*



- LRU maintenance budget: amount of LRU work during sleep
  - ➢ Empirically derived
  - ➢ Optimal budget *increases* with request rate (as LRU work increases)

Improvement in Application Latency

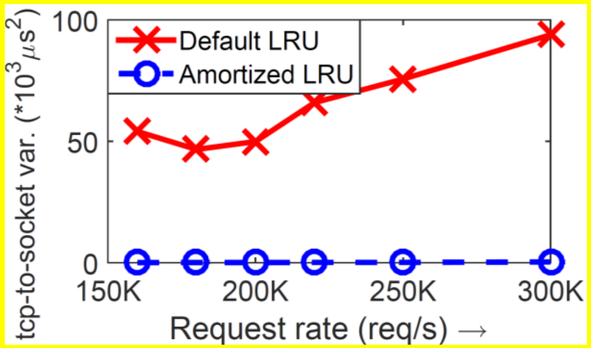


Constant load (300K req/s)

- Mean latency improves by about **20-28%**
- Tail latency improves by **4−32%**

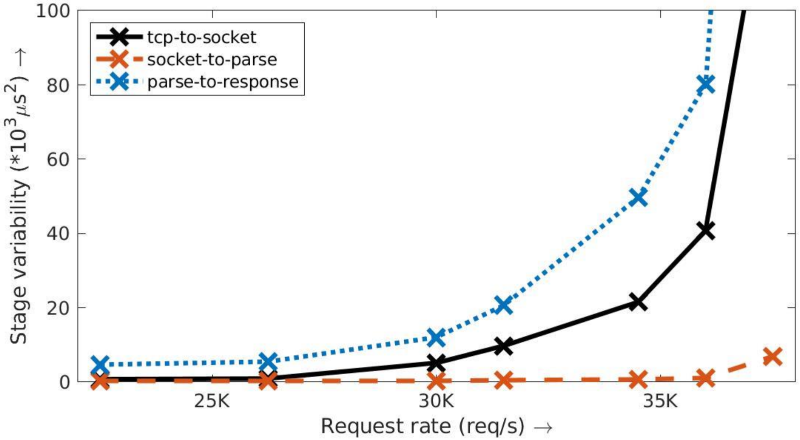
Facebook's VAR, APP, ETC traces

- Mean latency improvement: **22−31%**
- Tail latency improvement: **7−42%**



Application to Apache Web Server

- **Parse-to-response**: App processing
- **Tcp-to-socket**: Wakeup latency of app
  - ➢ Note: Variability increasing with req rate



Bottleneck analysis: Unpinned thread

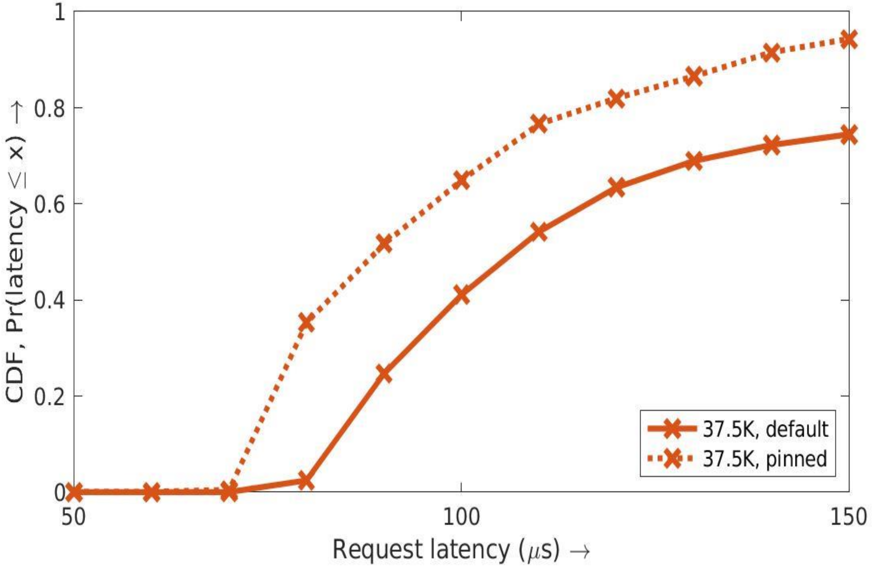
➢ Scheduled/awoken at request arrival

➢ Thread can be migrated, adds to variability, especially at high req rate

Control knob: Pin application threads, hopefully reduce thread migration variability

➢ Downside: Have to wait for pinned core, even if others are idle

Improvement in Application Latency



Constant load (37.5K req/s)

- Mean latency improvement: **15−50%**
- Tail latency improvement: **19−52%**

Facebook’s VAR, APP, ETC traces

- Mean latency improvement: **27−49%**
- Tail latency improvement: **36−62%**

Key Idea: Focus on Variability

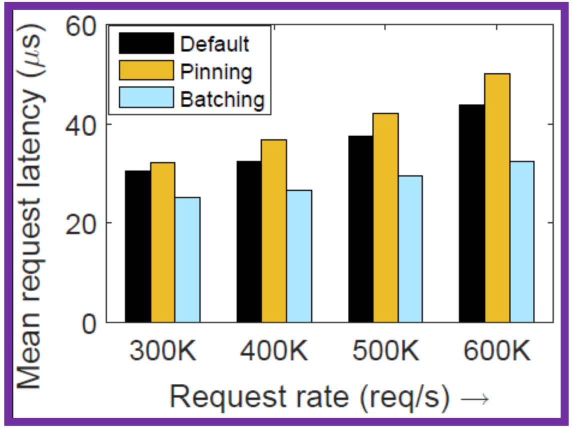
Using **variability of service time** for identifying bottleneck and control knob

Q1) What if we use mean service time (ST)?

- For Memcached low xput, mean ST suggests socket-to-parse
- But using optimal batching *hurts latency* by as much as 32%
- Variability of ST reduces latency by 30% by targeting tcp-to-socket (LRU idea)

Q2) What if we pick the wrong control knob?

- Memcached high xput: batching helps by 25%
- What if we use pinning?
- Pinning *hurts latency* by 12%



Limitations

- Request probing can add overhead
  - ➢ *As much as 5% in our case*

- Finding control knobs is not obvious
  - ➢ *Knobs may not generalize to other applications*
  - ➢ *Some ideas can generalize, e.g., focus on thread scheduling for tcp-to-socket*

- Control knobs require (empirical) tuning
  - ➢ *Not difficult, but requires offline work*

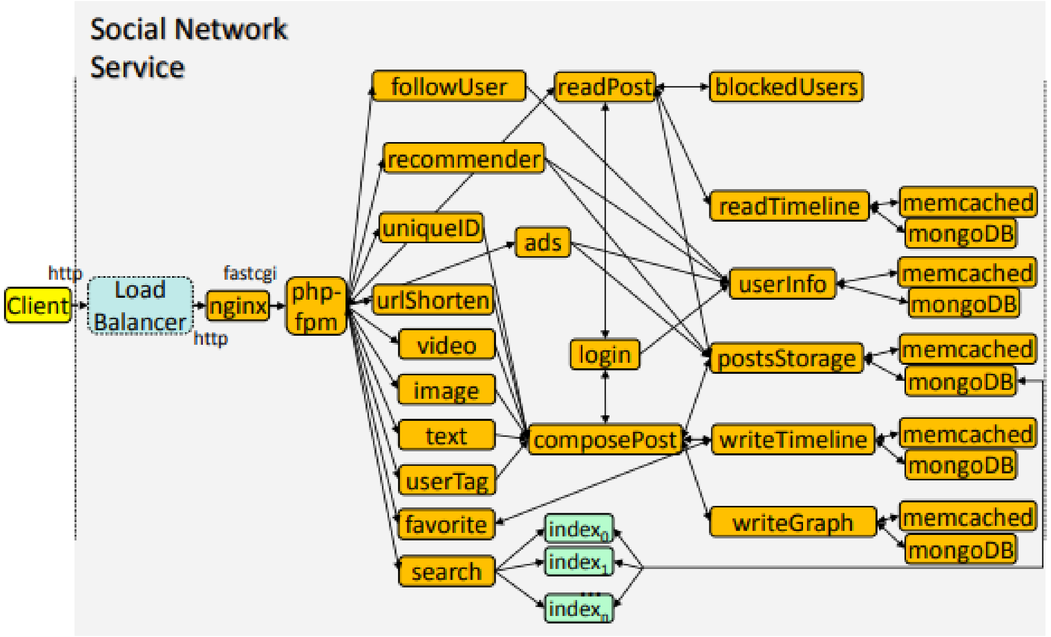
Outline

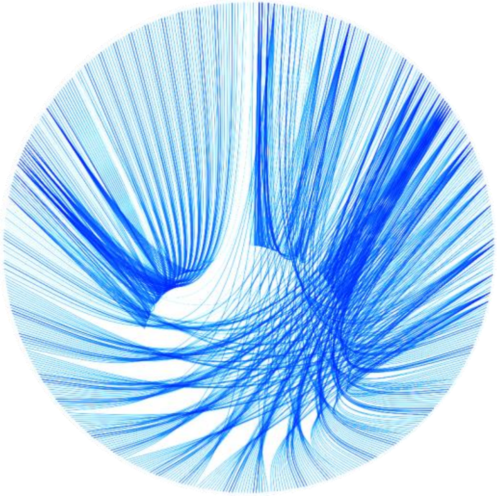
Part 2: Mitigating variability to reduce latency

- Application profiling: service time variability, stages of processing
- Control knobs: OS and application specific knobs to reduce variability
- Case studies: Memcached, Apache web server; alternative strategies
- Future work: multi-server, VMs, microservices
- Conclusion

Other Applications: Debugging Microservices

Microservices have 10s to 100s of services composing an application

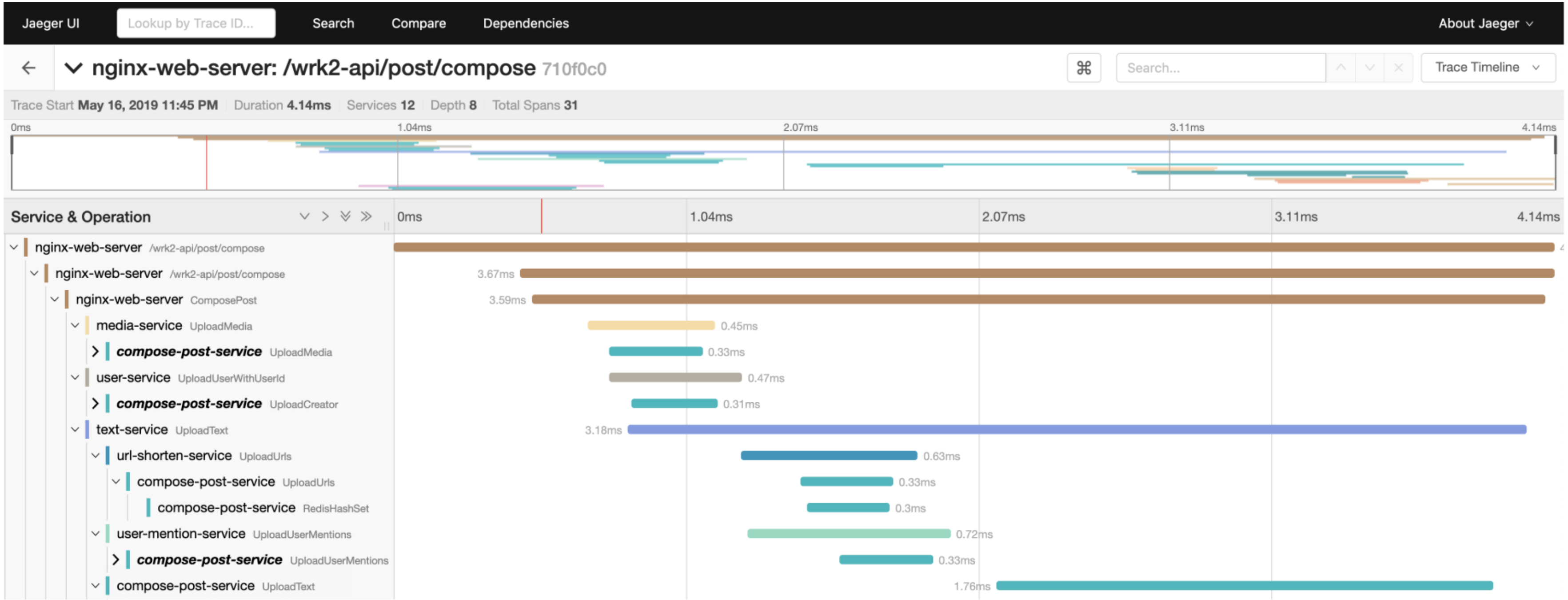




Typical stress points: Network processing, Scheduling Delays

Profiling Microservices

Build upon existing tracing infrastructure such as Jaeger for stage level breakdown



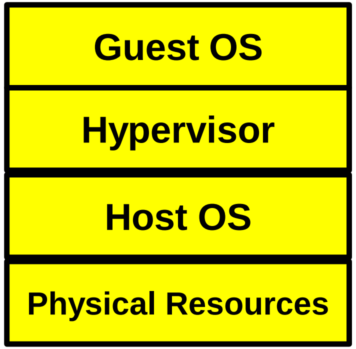
nginx-web-server: /wrk2-api/post/compose 710f0c0

Trace Start **May 16, 2019 11:45 PM** Duration **4.14ms** Services **12** Depth **8** Total Spans **31**

```
nginx-web-server /wrk2-api/post/compose
  nginx-web-server /wrk2-api/post/compose                 3.67ms
    nginx-web-server ComposePost                          3.59ms
      media-service UploadMedia                           0.45ms
        compose-post-service UploadMedia                  0.33ms
      user-service UploadUserWithUserId                   0.47ms
        compose-post-service UploadCreator                0.31ms
      text-service UploadText                             3.18ms
        url-shorten-service UploadUrls                    0.63ms
          compose-post-service UploadUrls                 0.33ms
            compose-post-service RedisHashSet             0.3ms
        user-mention-service UploadUserMentions           0.72ms
          compose-post-service UploadUserMentions         0.33ms
        compose-post-service UploadText                   1.76ms
```

Other Applications: Multi-tier VM deployments

- Cloud hosted web applications use multi-tier VM setups
- VM relies on the guest OS, hypervisor, and host OS, to get access to physical resources.
- Need to probe multiple abstractions – guest OS, hypervisor, host OS.
- The timestamps collected in this case (hypervisor and guest OS) will be passed back to the host OS



Outline

Part 2: Mitigating variability to reduce latency

- Application profiling: service time variability, stages of processing
- Control knobs: OS and application specific knobs to reduce variability
- Case studies: Memcached, Apache web server; alternative strategies
- Future work: multi-server, VMs, microservices
- Conclusion

Conclusion

- Presented an approach, inspired by QT, to find/mitigate latency bottlenecks
  - Memcached
    - High-xput (bounded batching): Mean-latency: **24-26%**, Tail latency: **34-40%**
    - Low-xput (LRU amortization): Mean-latency: **20-28%**, Tail latency: **4-32%**
  - Apache Web server
    - (thread pinning): Mean-latency: **15-50%**, Tail latency: **19-52%**

Variability as a guiding principle for system design

Thank you!

Anshul Gandhi and Amoghavarsha Suresh

PACE

Stony Brook University

Backup Slides