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.

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
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
Significance of Variability

Variability represents an opportunity for reducing latency
Goal of this Tutorial

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 balancing
- 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 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
Queueing Theory Origins

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

PDF Conversion/Compliance

Please be patient, this process may take a few minutes. Mr.
Every 7 seconds this page will refresh to check the status of
You can check the latest status by clicking on the following
You can cancel this process by clicking the following link:
Popular Applications of Queueing Theory

Standard Shipping

Order → Fulfillment center → U.S. Postal Service → You

Same Day

Order → Fulfillment center → Sorting center → U.S. Postal Service → You

Prime Now

Order → Store → Crowdsourced delivery → You
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, \( \text{IAT} \), time between requests
    ➢ \( 1/E[\text{IAT}] = \lambda \) requests/sec (average arrival rate)
  ➢ Service time, \( \text{ST} \), size of a request
    ➢ \( 1/E[\text{ST}] = \mu \) requests/sec (average service rate)

• Can also model number of requests in system \( (N) \) or queue \( (N_Q) \)
Arrivals and Services

- $1/E[IAT] = \lambda$ requests/sec (average arrival rate)
- $1/E[ST] = \mu$ requests/sec (average service rate)

- Assume $\lambda < \mu$ always
- Why? What if $\lambda > \mu$??

- 4 GHz server
- Single-threaded CPU-intensive job requiring 1 Gigacycles to complete
- $E[ST] = ??$ seconds
- $\mu = ?$ req/s
1/E[IAT] = \( \lambda \) requests/sec (average arrival rate)

1/E[ST] = \( \mu \) requests/sec (average service rate)

- Average incoming work/sec

- Note, \( \rho < 1 \)

Load (\( \rho \)) = E[ST]/E[IAT] = \( \lambda / \mu \)

- \( \mu = 4 \text{ req/s} \)
- \( \lambda = 2 \text{ req/s} \)
- \( \rho = ?? \)
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
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??

![Diagram showing λ and μ with an arrow from λ to μ](image)
Throughput is **average rate at which requests are serviced**

- Avg. arrival rate \( \lambda \) req/s
- Avg. service rate \( \mu \) req/s
- Assume no losses

**Throughput** = ??

**Peak throughput** = ??

\[
\lambda \quad \rightarrow \quad 2\mu
\]
In Practice: Arrivals and Services

- \( \lambda \) and \( \mu \) are key parameters of queueing models
- But how to obtain these in practice? Not always readily available.

2. \( \mu \) is average service rate

- \( \mu \) is same as peak throughput
- \( \mu = \frac{1}{E[ST]} \)
- \( ST \): time to service request (no queueing)
- Measure \( E[ST] \) and set \( \mu = \frac{1}{E[ST]} \)
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

  ![Queueing Model Diagram](image-url)
Significance of the IAT and ST Distribution

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

$E[ST] = 1 \text{ ms; } E[IAT] = \text{Load}/E[ST]$
Significance of the IAT and ST Distribution

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

\[ E[ST] = 1 \text{ ms}; \quad E[IAT] = \text{Load}/E[ST] \]

\[ \text{M/D/1 model} \]
Common distributions:

- **D**: Deterministic (zero var)
- **M**: Exponential (medium var)

\[
f(x) = \frac{1}{e^x}
\]

\[\text{Mean} = 2\]
IAT and ST Distributions

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

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

Mean = 2
IAT and ST Distributions

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

\[ f(x) = \frac{1}{x^{\alpha+1}} \]

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

Mean = 2

"Heavy" tail
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 $ST$

  ![Queueing Model Diagram]

- 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

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
- $M/M/1: \text{E}[T] = 1/(\mu - \lambda) = \text{E}[ST]/(1 - \rho)$
- $T_{95} = \text{E}[ST] \times \text{ln}(20)/(1 - \rho)$
- $T_x = \text{E}[ST] \times \text{ln}(1 -.01x)/(1 - \rho)$

**Takeaway 1**
Latency $\sim 1 /(1 - \rho)$

$E[ST] = 1 \text{ ms}; \quad E[IAT] = \text{Load}/E[ST]$
Queueing Models: Results

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

• For a given load:
  \[ T_{M/H_2/1} > T_{M/M/1} > T_{M/D/1} > T_{D/D/1} \]

Takeaway 2

Latency increases with load and IAT and ST variability
In Practice, latency $\sim 1/(1 - \rho)$, and not latency $\sim \rho$

However, in practice, latency $\neq E[ST]/(1 - \rho)$

- 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:

\[ T = \alpha_1 + \frac{1}{(1 - \alpha_2 \rho)^{\alpha_3}} \]

Solve for \( \alpha \) via regression or control theory

---

In Practice: Queueing Models

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

Takeaway 3

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

Takeaway 4

The H2 distribution can be tuned via its parameters to provide an adequate fit for IAT and ST
Today’s applications employ a cluster of servers to serve the workload.

Queueing model: \( D_{\text{IAT}} / D_{\text{ST}} / k \) model

**Scheduling:** idle server picks request from head-of-queue
Multi-Server Queueing Models: Results

- M/M/k

**Takeaway 5**

- \( \Pr(\text{all } k \text{ servers busy}) \sim \rho^k \)
- With more servers, we can better handle load variations

\[ \rho = \frac{\lambda}{k\mu} < 1 \]
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

queue with (blue) requests

$p_k$
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**

\[
p_i^* = \frac{\mu_i \cdot \sum_j \sqrt{\mu_j} - \sqrt{\mu_i} \cdot \sum_j \mu_j + \lambda \cdot \sqrt{\mu_i}}{(\lambda \cdot \sum_j \sqrt{\mu_j})}
\]

S. Javadi et al., DIAL, ICAC 2017
A. Gandhi et al., HALO, Mascots 2015
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)
- **H₂**: Hyper-exponential (high var)

- Service time, \( ST \), size of a request
  - \( \text{Var}(ST) \) is important
  - But what about \( E[ST] \) ?

\[
E[ST] = 1 \text{ ms}; \quad E[IAT] = \frac{\text{Load}}{E[ST]}
\]
Impact of $\text{Var(} \text{ST}) \text{ and } E[\text{ST}]$ on Latency

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

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

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

![Latency heatmap as function of $\text{Var(} \text{ST}) \text{ and } E[\text{ST}]$](image)

**Takeaway 7**

Reducing $\text{Var(} \text{ST})$, even at the expense of $E[\text{ST}]$, can significantly reduce latency.
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

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

Takeaway 1
Latency $\sim 1 / (1 - \rho)$

Takeaway 2
Latency increases with load and IAT and ST variability

Takeaway 3
$$T = \alpha_1 + \frac{1}{(1 - \alpha_2 \cdot \rho)^\alpha},$$

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) $\sim \rho^k$
- With more servers, we can better handle load variations

Takeaway 6
$$p_i^* = \frac{\mu_i \cdot \sum_j \sqrt{\mu_j} - \sqrt{\mu_i} \cdot \sum_j \mu_j + \lambda \cdot \sqrt{\mu_i}}{\left(\lambda \cdot \sum_j \sqrt{\mu_j}\right)}$$

Takeaway 7
Reducing $\text{Var(ST)}$, even at the expense of $E[ST]$, can significantly reduce latency
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
PB0: Network stack processing

PB1: Worker thread wakeup

do_sock_read()

PB2: Batch of requests arrive

PB3: App processing

sock_sendmsg()

PB4: App processing

TCP enqueues to socket

NIC driver

App reads at socket

App begins processing

App ends processing
Computing Variability of Service Time at Each Stage

• $\text{Var}(S) = \text{E}[S^2] - (\text{E}[S])^2$

  ➢ $\text{E}[S] \approx (s_1 + s_2 + \ldots + s_n)/n$; $\text{E}[S^2] \approx (s_1^2 + s_2^2 + \ldots + s_n^2)/n$

    ▪ $n$ requests
    ▪ $s_i$: duration for request $i$

  ➢ Only need running sum of duration ($S$) and its square ($S^2$)

  ➢ Low overhead
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: $(x_{w+1} - x_1)(x_w - \mu_{w+1} + x_1 - \mu_w)$
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 $\rightarrow$ batch size variability

[Graph showing CDF of batch size distribution]
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
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*
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
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
Backup Slides