LOAD BALANCING
IS
IMPOSSIBLE

Tyler McMullen

tyler@fastly.com

@tbmcmullen

WHAT IS LOAD BALANCING?

[DIAGRAM DESCRIBING LOAD BALANCING]

[ALLEGORY DESCRIBING LOAD BALANCING]

Why Load Balance?

Three major reasons. The least of which is balancing load.

Abstraction

Treat many servers as one

Single entry point

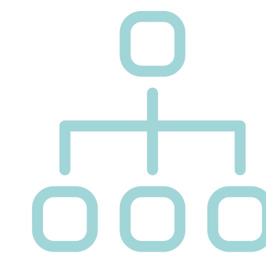
Simplification

Failure

Transparent failover

Recover seamlessly

Simplification



Balancing Load

Spread the load efficiently across servers

RANDOM

THE INGLORIOUS DEFAULT

AND BANE OF MY EXISTENCE

What's good about random?

- Simplicity
- Few edge cases
- Easy failover
- Works identically when distributed

What's bad about random?

- Latency
- Especially long-tail latency
- Useable capacity

BALLS-INTO-BINS

If you throw m balls into n bins,
what is the maximum load
of any one bin?

Theorem 1. *Let M be the random variable that counts the maximum number*

$$k_\alpha = \begin{cases} \frac{\log n}{\log \frac{n \log n}{m}} \left(1 + \alpha \frac{\log^{(2)} \frac{n \log n}{m}}{\log \frac{n \log n}{m}}\right), & \text{if } \frac{n}{\text{polylog}(n)} \le m \ll n \log n, \\ (d_c - 1 + \alpha) \log n, & \text{if } m = c \cdot n \log n \text{ for some constant } c, \\ \frac{m}{n} + \alpha \sqrt{2 \frac{m}{n} \log n}, & \text{if } n \log n \ll m \le n \cdot \text{polylog}(n), \\ \frac{m}{n} + \sqrt{\frac{2m \log n}{n} \left(1 - \frac{1}{\alpha} \frac{\log^{(2)} n}{2 \log n}\right)}, & \text{if } m \gg n \cdot (\log n)^3. \end{cases}$$

The paper is organized as follows: in § 2 we give a brief overview of the first and second moment method, in § 3 we show how to apply this method within the balls-into-bins scenario and obtain in § 4 the $\frac{\log n}{\log \log n}(1 + o(1))$ bound for $m = n$. In § 5 we then present some more general tail bounds for Binomial random variables and combine them with the first and second moment method to obtain a proof of Theorem 1.

```python
import numpy as np
import numpy.random as nr

n = 8    # number of servers
m = 1000 # number of requests

bins = [0] * n

for chosen_bin in nr.randint(0, n, m):
    bins[chosen_bin] += 1

print bins
```

[129, 100, 134, 113, 117, 136, 148, 123]

```python
import numpy as np
import numpy.random as nr

n = 8    # number of servers
m = 1000 # number of requests

bins = [0] * n

for weight in nr.uniform(0, 2, m):
    chosen_bin = nr.randint(0, n)
    bins[chosen_bin] += weight

print bins
```

[133.1, 133.9, 144.7, 124.1, 102.9, 125.4, 114.2, 121.3]

How do you model request latency?

What do

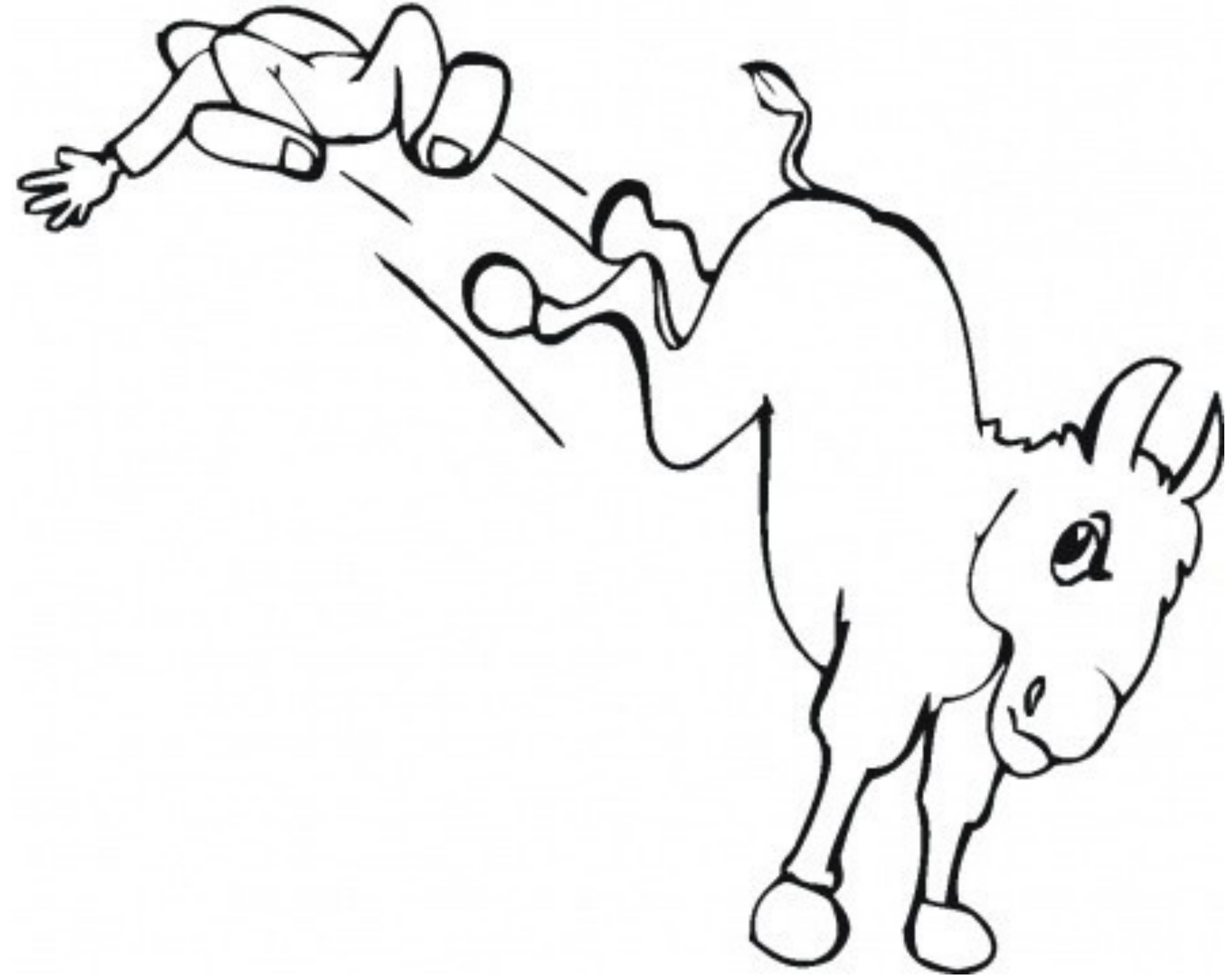
Erlang

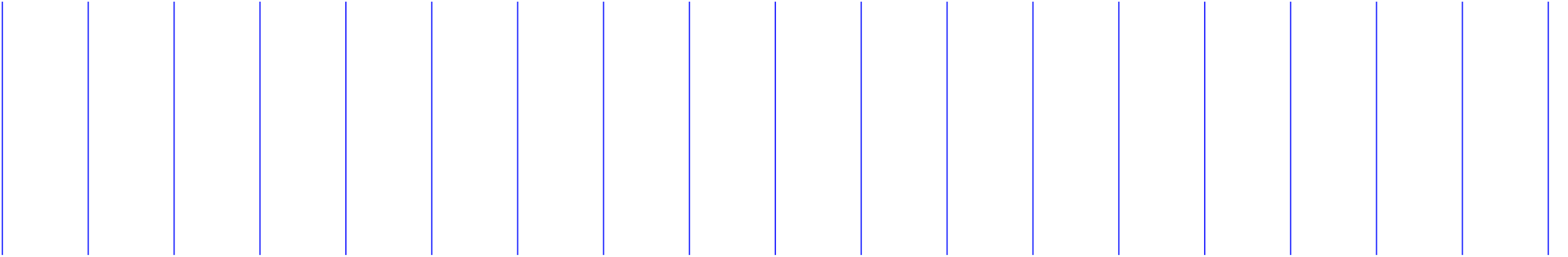
and

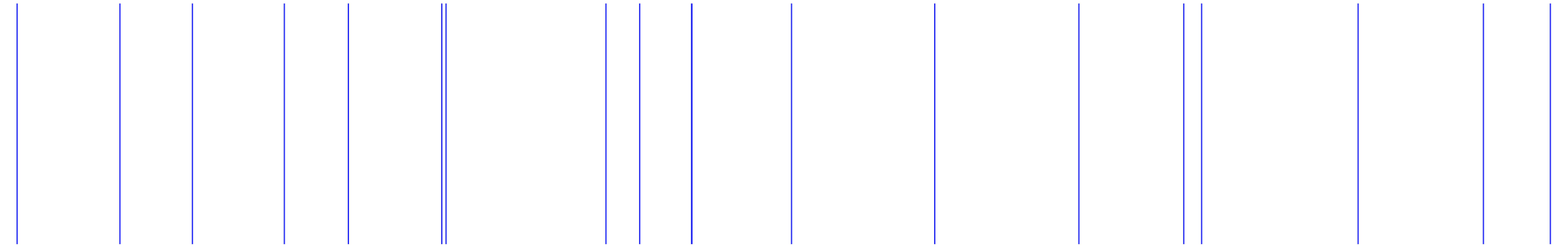
getting kicked by a horse

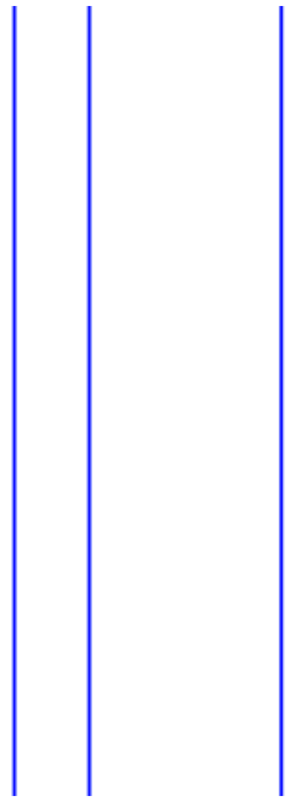
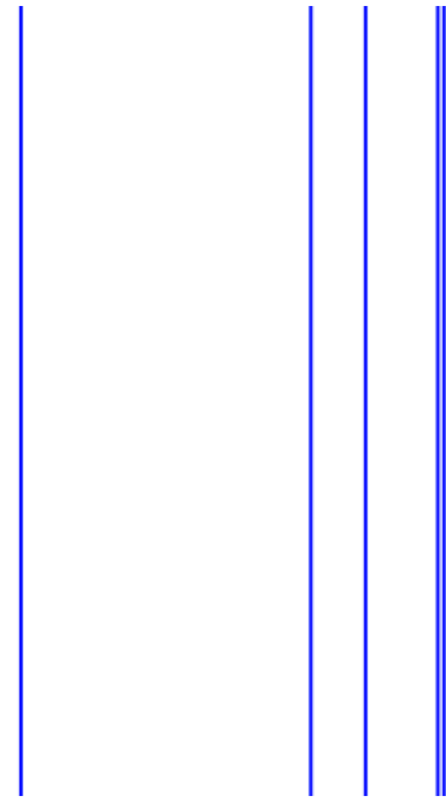
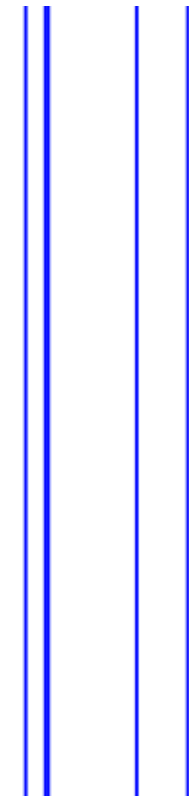
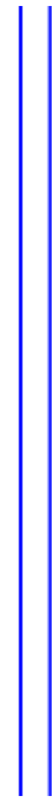
have in common?

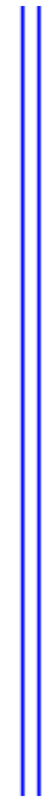
POISSON PROCESS



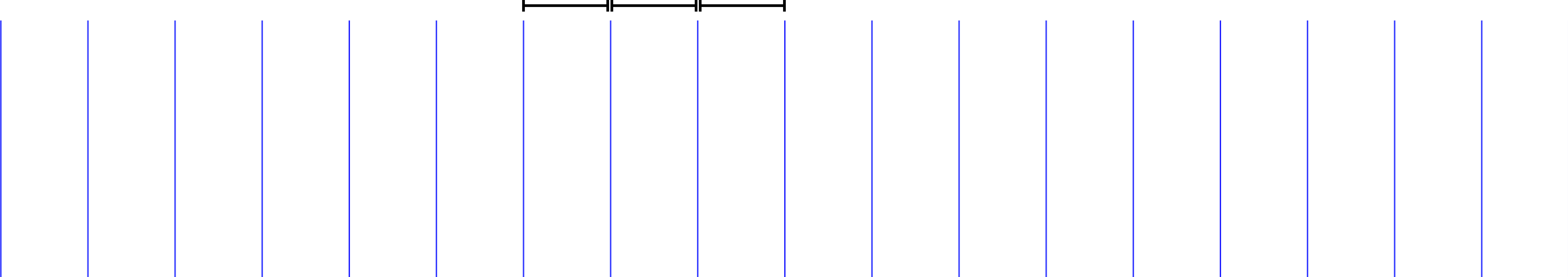


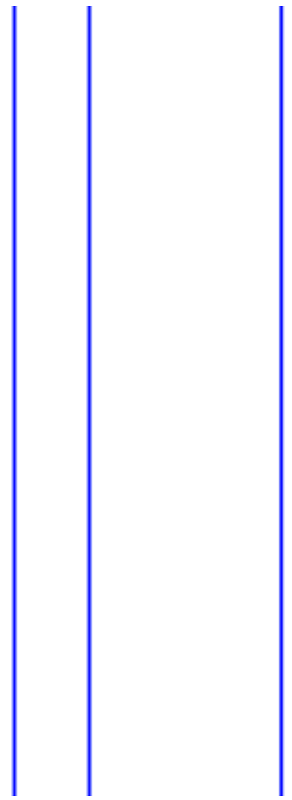
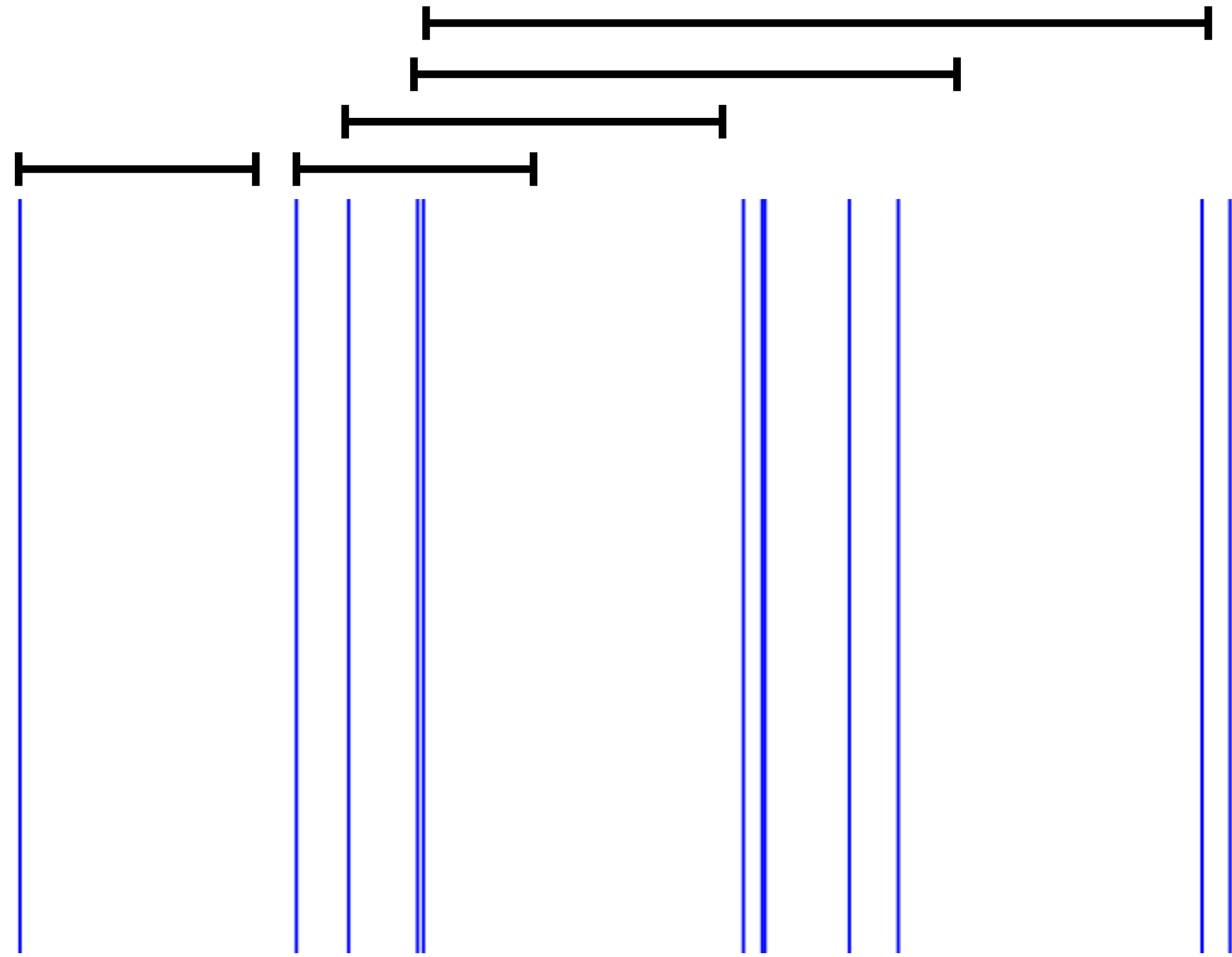
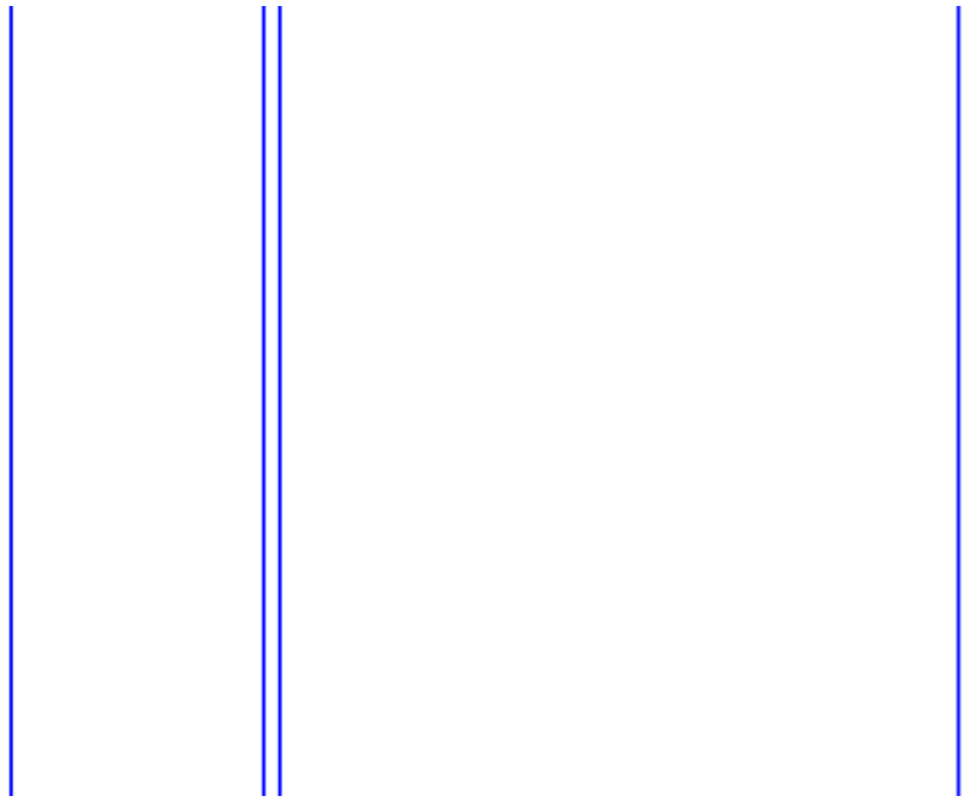




WHY IS THAT A PROBLEM?


50ms

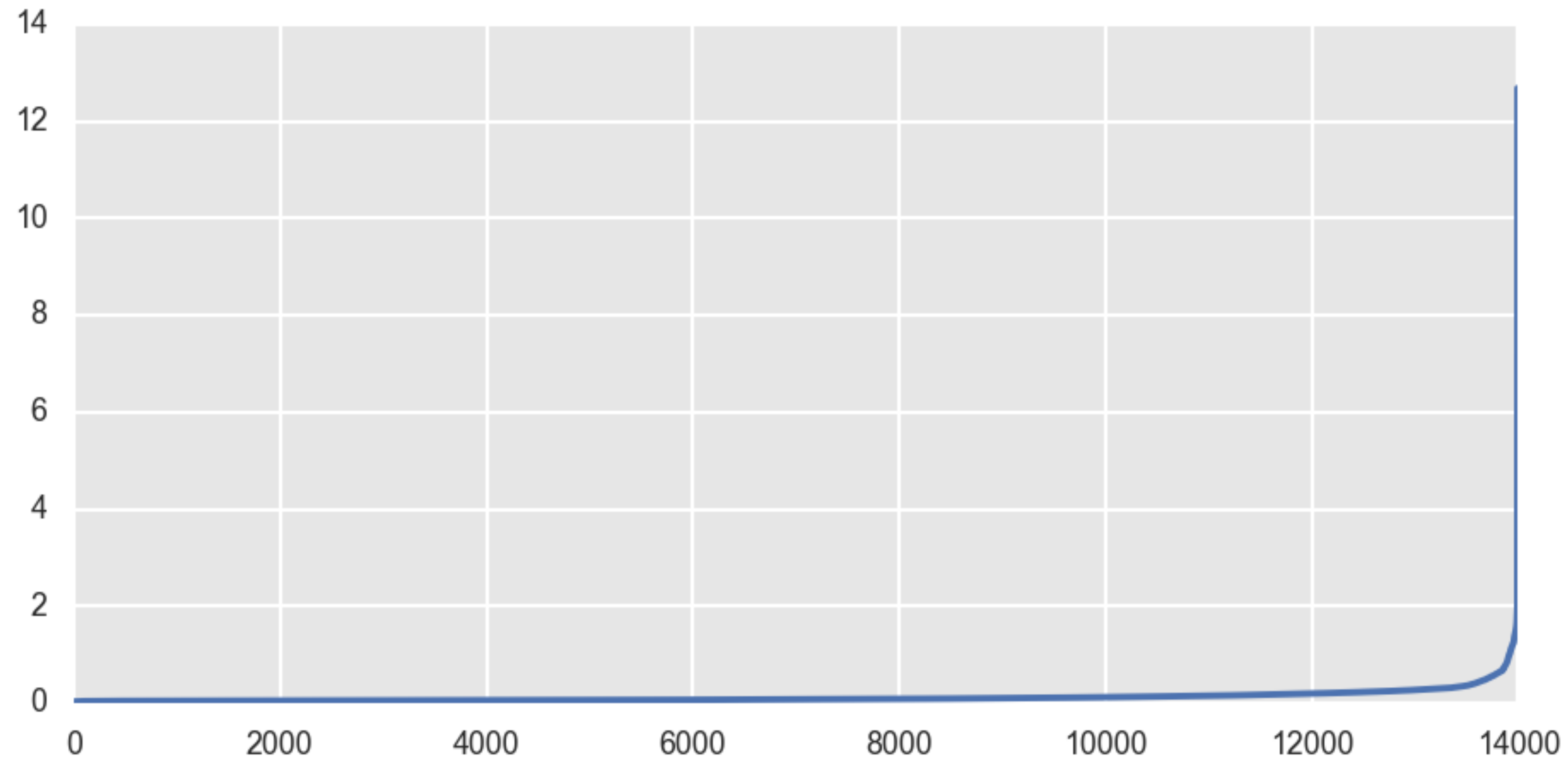




Even if your application has perfect constant response time

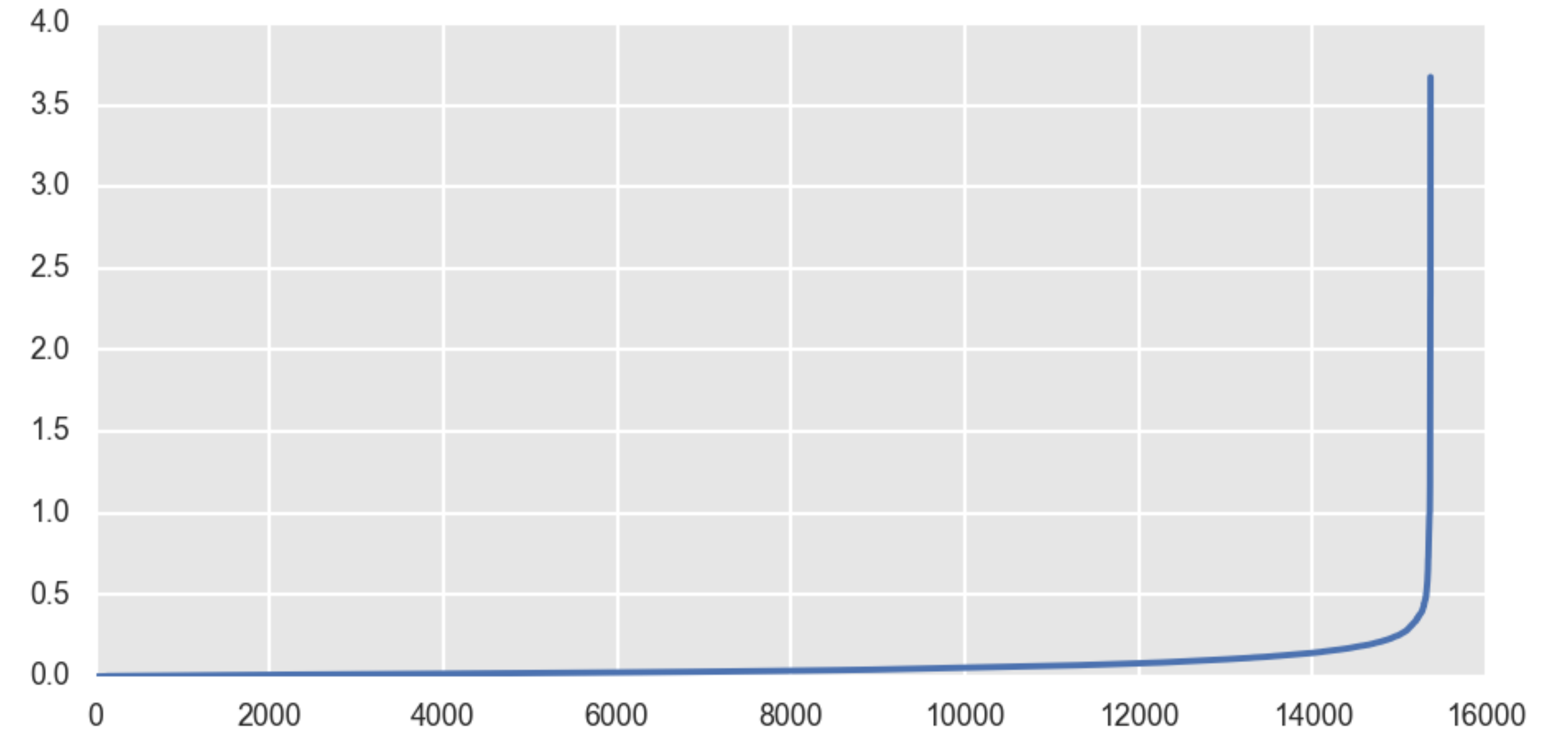
... It doesn’t.

Log-normal Distribution

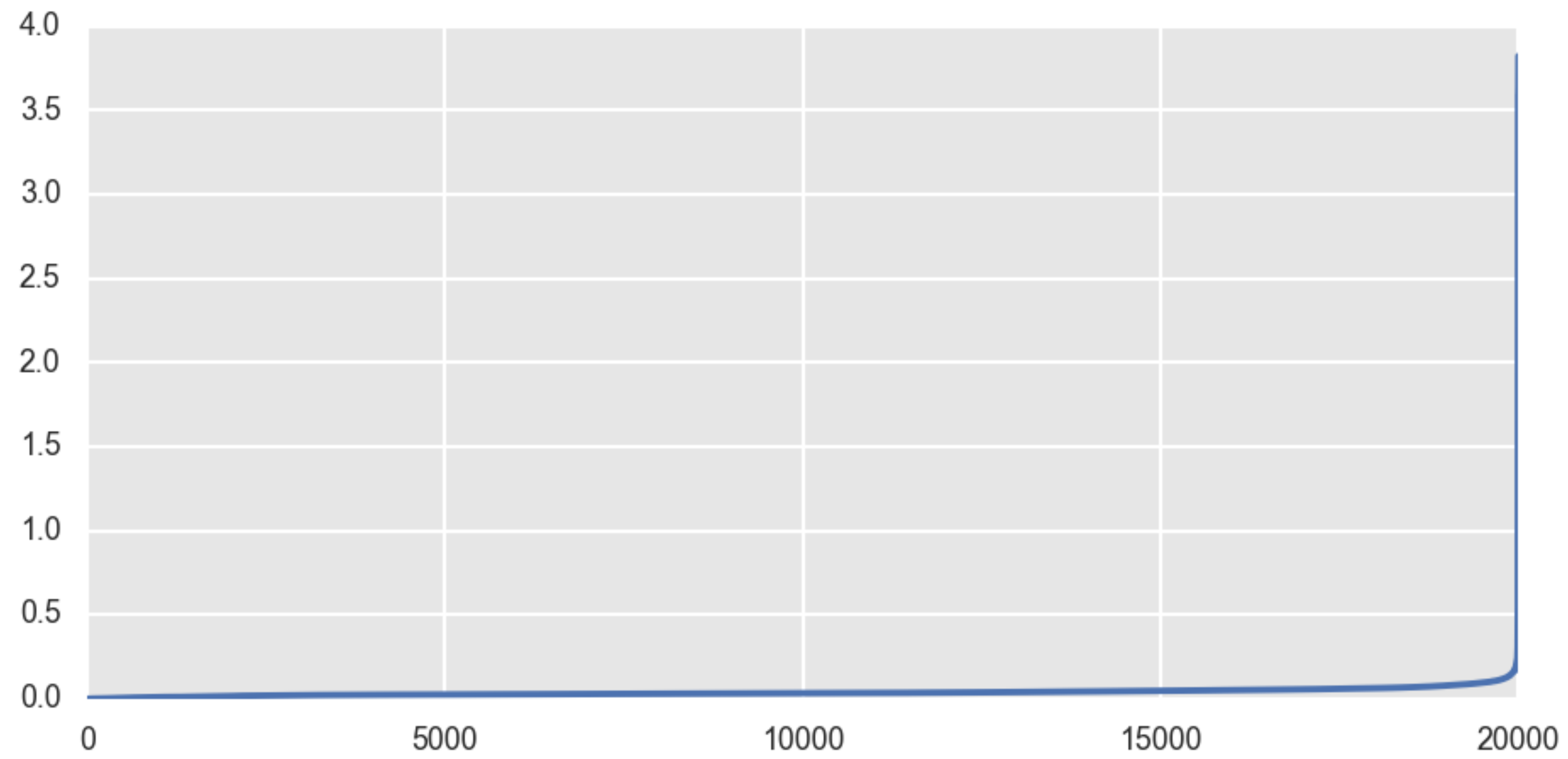
User-Generated Content



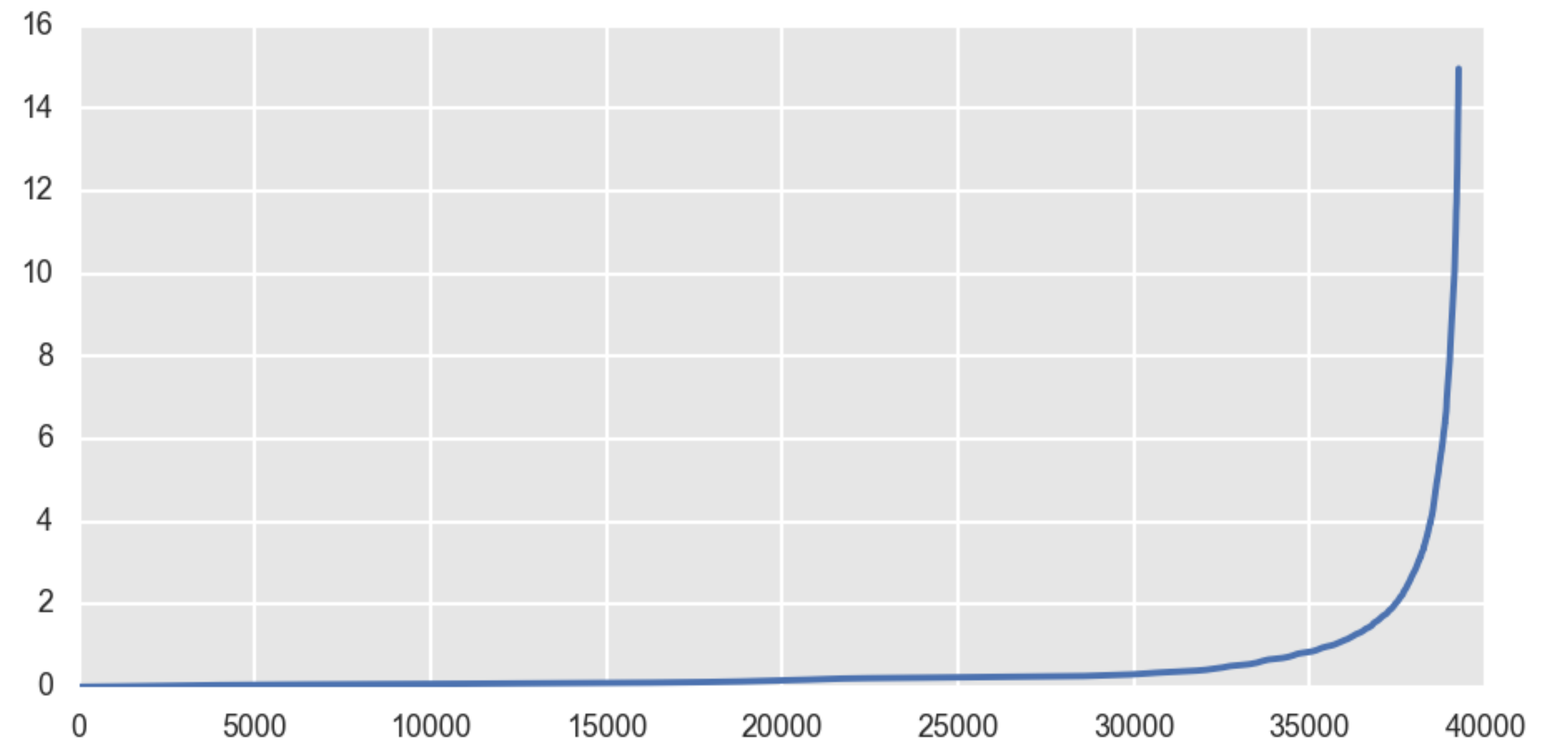
Social



Ad-serving



Photos



```
mu = 0.0
sigma = 1.15
lognorm_mean = math.e ** (mu + sigma ** 2 / 2)

desired_mean = 1.0

def normalize(value):
    return value / lognorm_mean * desired_mean

for weight in nr.lognormal(mu, sigma, m):
    chosen_bin = nr.randint(0, n)
    bins[chosen_bin] += normalize(weight)
```

[128.7, 116.7, 136.1, 153.1, 98.2, 89.1, 125.4, 130.4]

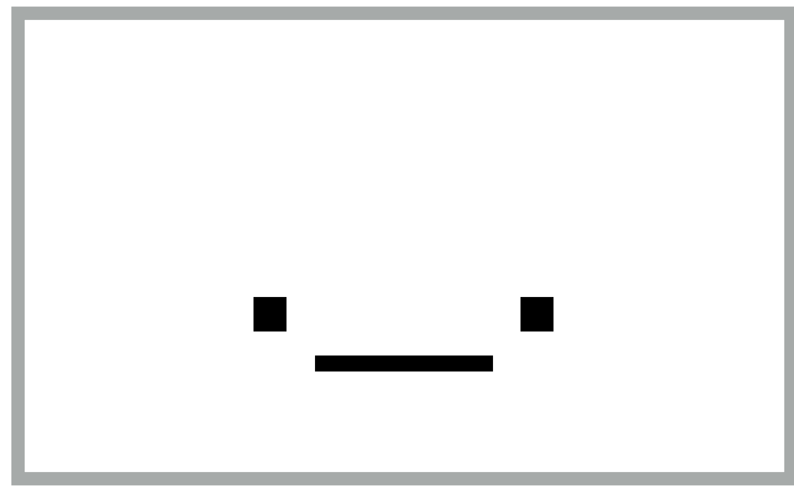
```python
mu = 0.0
sigma = 1.15
lognorm_mean = math.e ** (mu + sigma ** 2 / 2)

desired_mean = 1.0
baseline = 0.05

def normalize(value):
    return (value / lognorm_mean
            * (desired_mean - baseline)
            + baseline)

for weight in nr.lognormal(mu, sigma, m):
    chosen_bin = nr.randint(0, n)
    bins[chosen_bin] += normalize(weight)
```

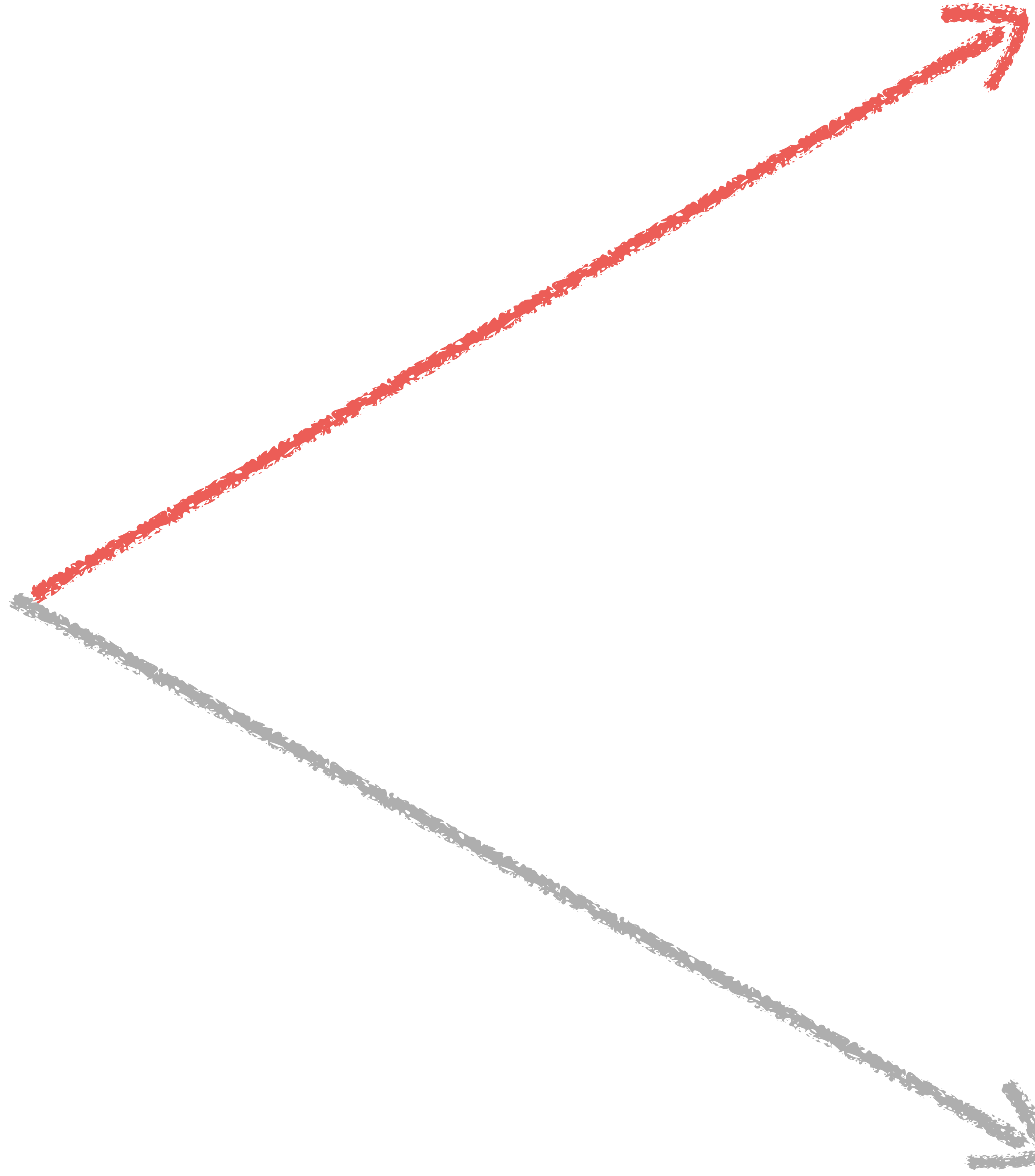
[100.7, 137.5, 134.3, 126.2, 113.5, 175.7, 101.6, 113.7]

THIS IS WHY

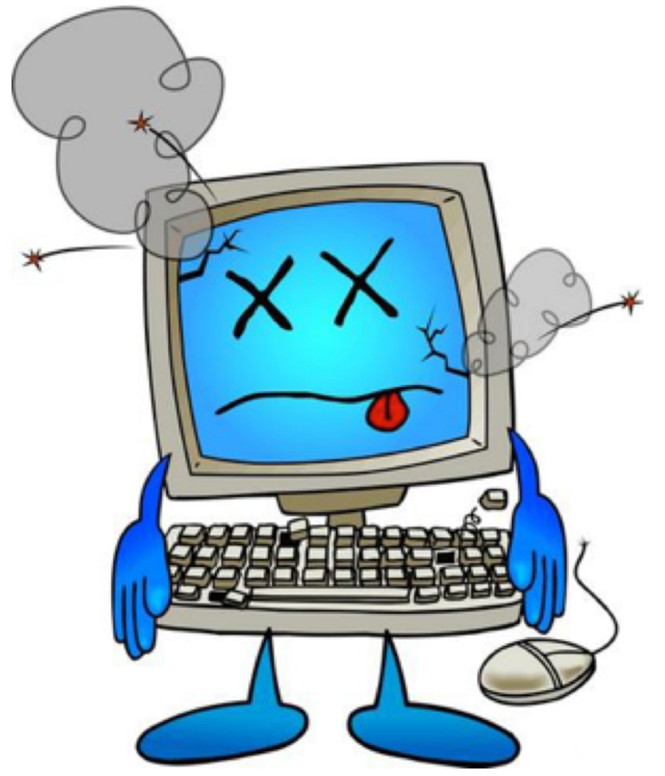
PERFECTION

IS IMPOSSIBLE





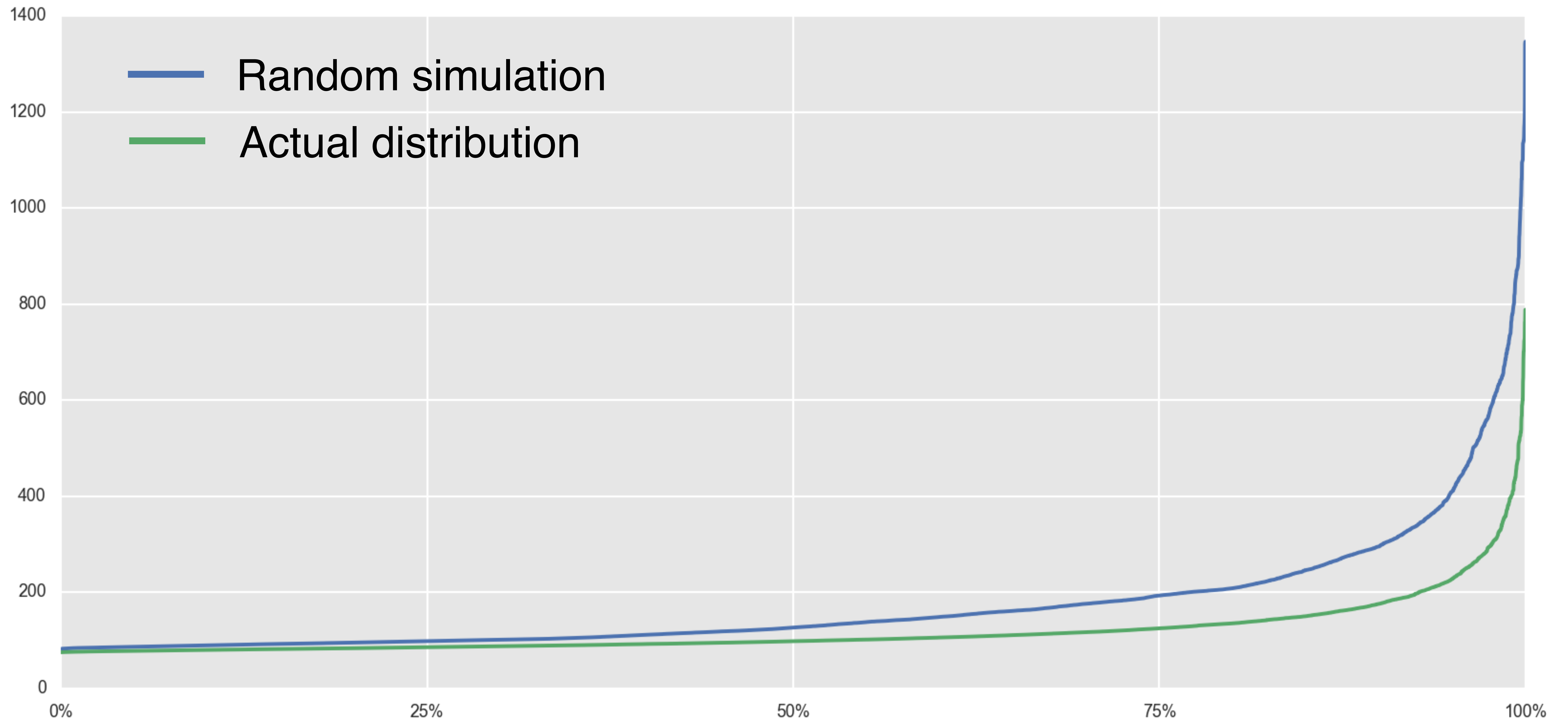
1



2

4

WHAT EFFECT DOES IT HAVE?


Random simulation
Actual distribution
1400
1200
1000
800
600
400
200
0
0%
25%
50%
75%
100%

Latency Tip Of The Day: #L

latencytipoftheday.blogspot.co.za/2014/06/latencytipoftheday-most-page-loads.html

More Next Blog» Create Blog Sign In

The probability of a single resource request avoiding the 99th percentile is 99%.

The probability of all *N* resource requests in a page avoiding the 99th percentile is (99% ^ *N*).

99% ^ 69 = 49.9%

About Me

Gil Tene

CTO and co-founder of Azul Systems.

View my complete profile

Blog Archive

▼ 2014 (8)

▼ June (8)

#LatencyTipOfTheDay: Median Server Response Time: ...

#LatencyTipOfTheDay: MOST page loads will experien...

#LatencyTipOfTheDay: Q: What's wrong with this pic...

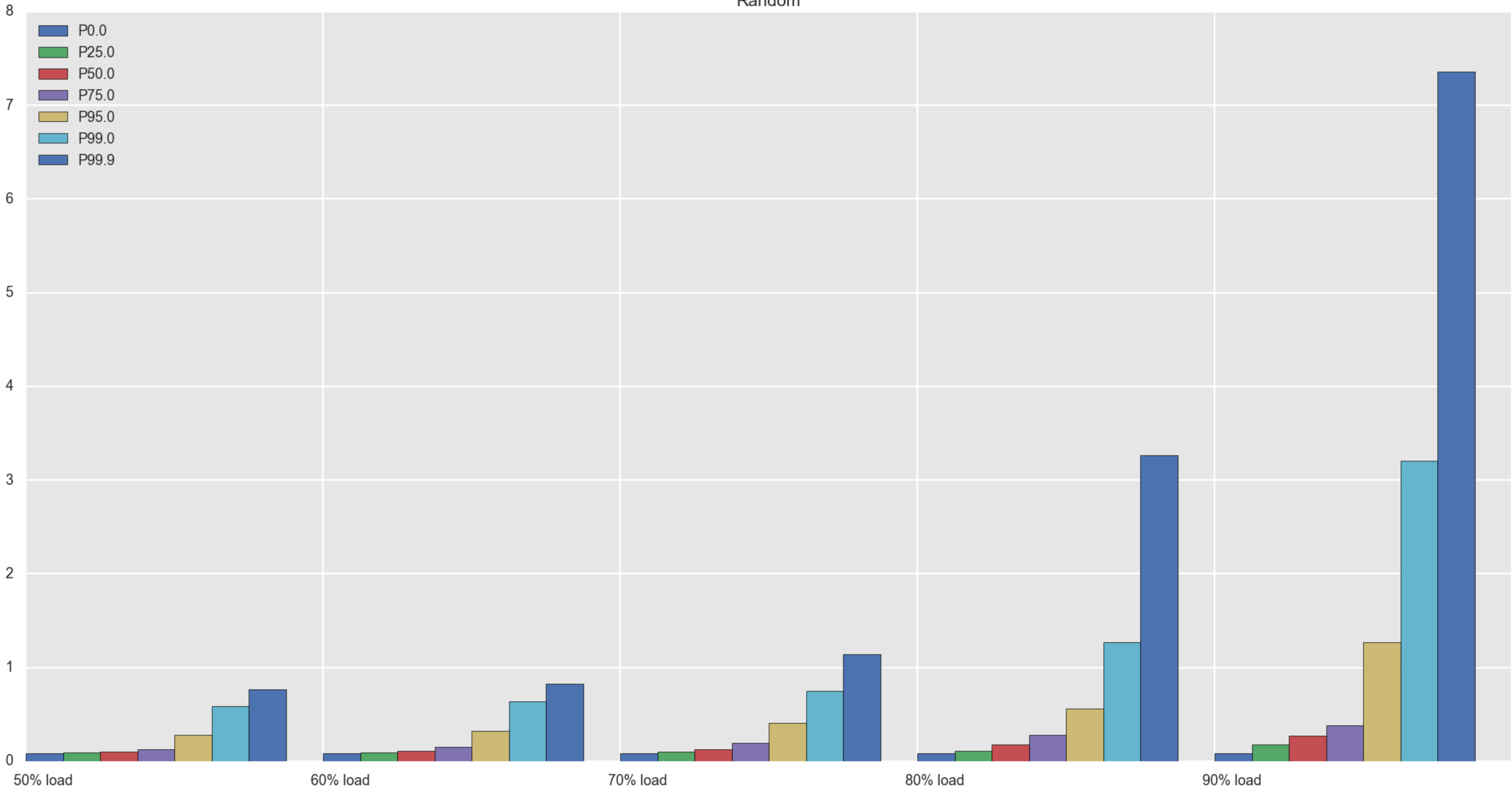
#LatencyTipOfTheDay: If you are not measuring and/...

#LatencyTipOfTheDay :

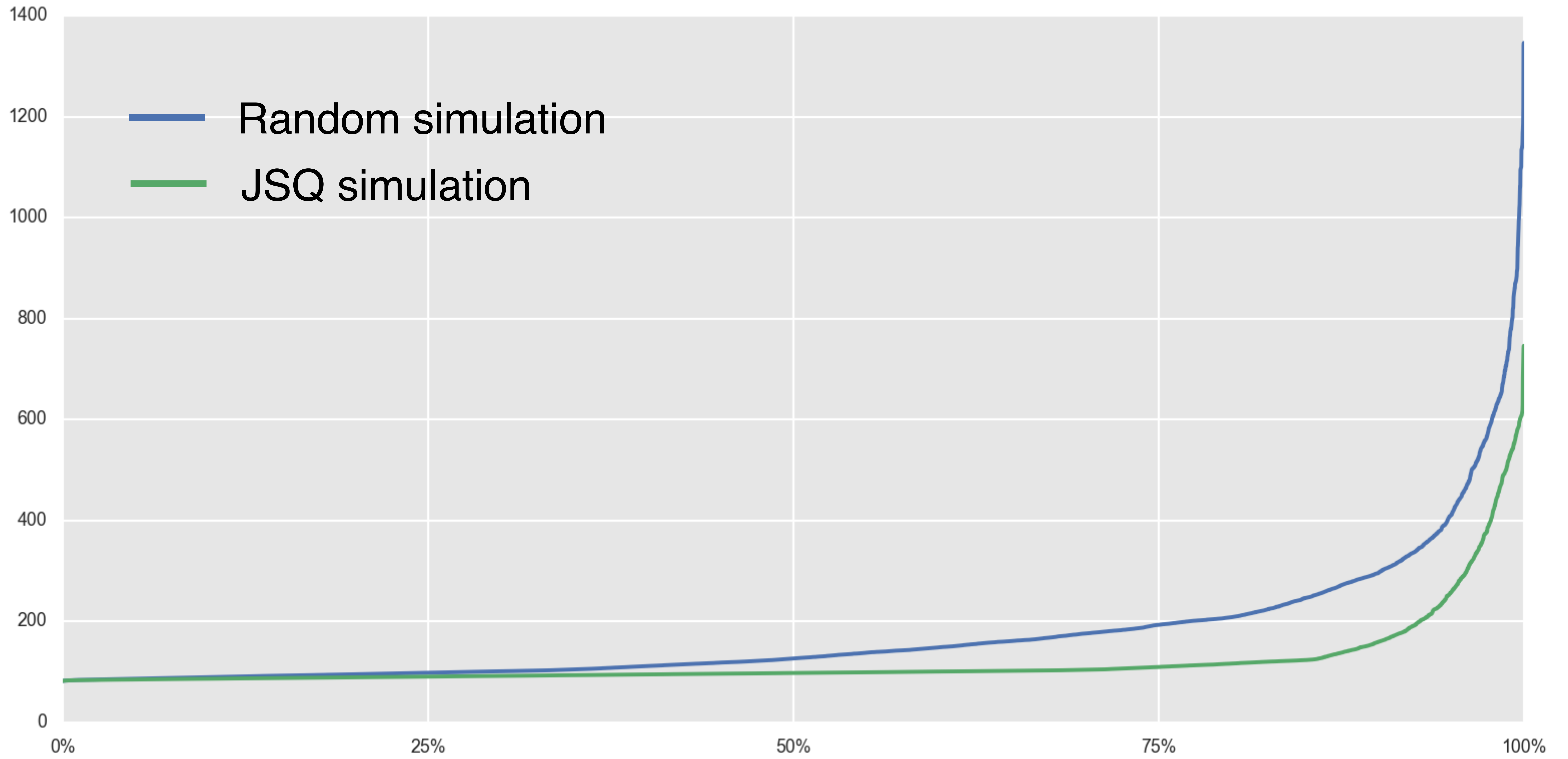
... page loads will experience the 99%'lie server response

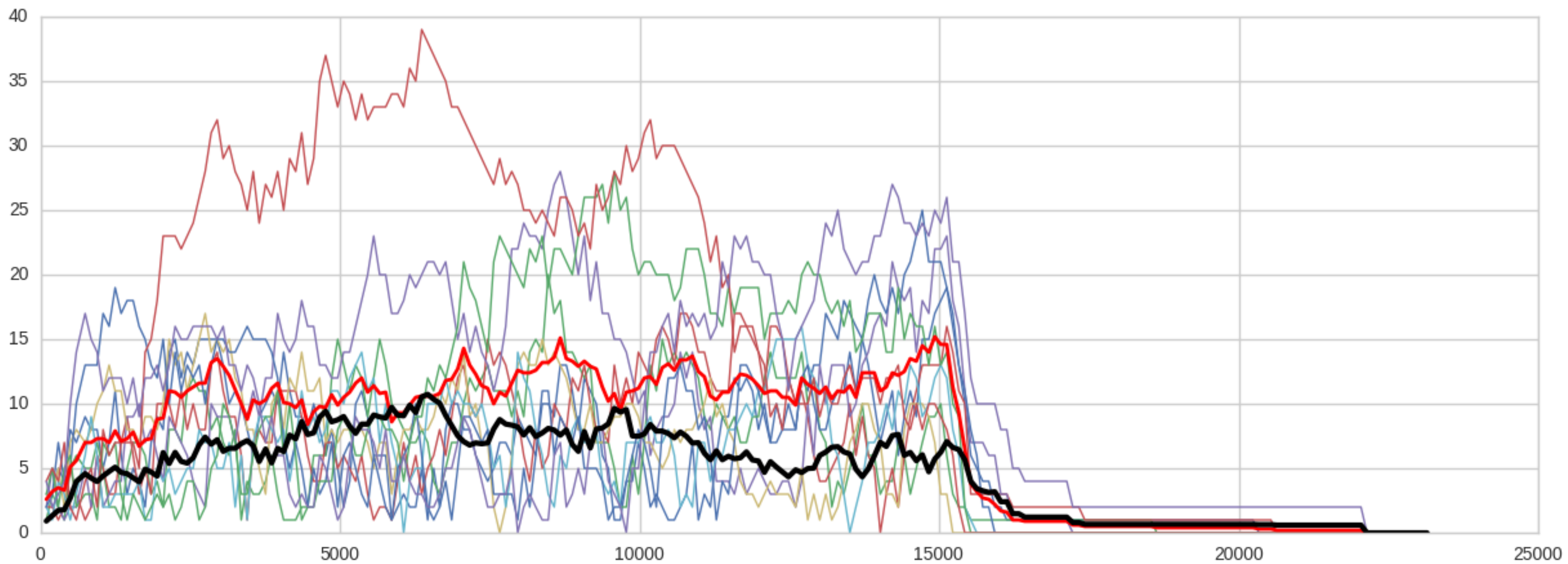
Yes. **MOST** of the page view attempts will experience the 99%'lie server response time in modern web applications. You didn't read that wrong.

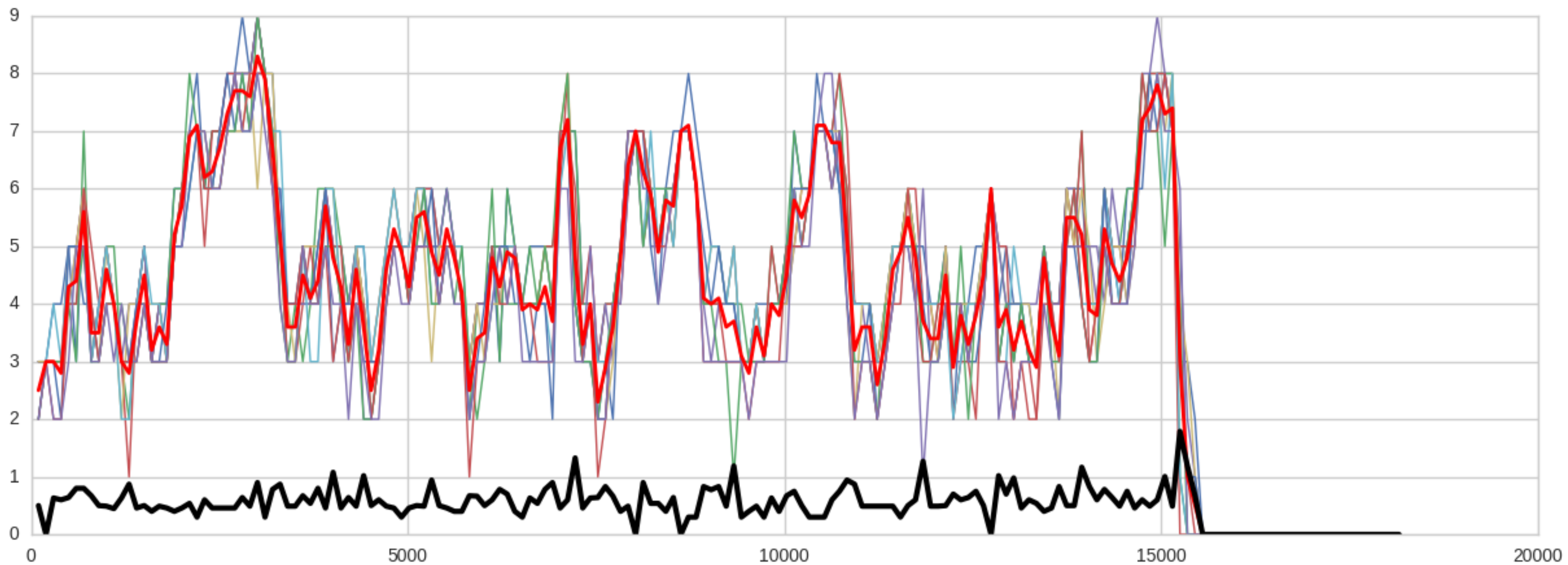
This simple fact seems to surprise many people. Especially people who spend much of their time looking at pretty lines depicting averages, 50%'lie, 90%'lie or 95%'lies of server response time in feel-good monitoring charts. I am constantly amazed by how little attention is paid to the "higher end" of the percentile spectrum in most application monitoring, benchmarking, and tuning


Random
P0.0
P25.0
P50.0
P75.0
P95.0
P99.0
P99.9
0
1
2
3
4
5
6
7
8
50% load
60% load
70% load
80% load
90% load

SO WHAT DO WE DO ABOUT IT?


Random simulation
JSQ simulation
1400
1200
1000
800
600
400
200
0
0%
25%
50%
75%
100%


0
5
10
15
20
25
30
35
40
0
5000
10000
15000
20000
25000


9
8
7
6
5
4
3
2
1
0
0
5000
10000
15000
20000

Random

P0.0
P25.0
P50.0
P75.0
P95.0
P99.0
P99.9

8
7
6
5
4
3
2
1
0

50% load
60% load
70% load
80% load
90% load

Join-shortest-queue

8
7
6
5
4
3
2
1
0

50% load
60% load
70% load
80% load
90% load

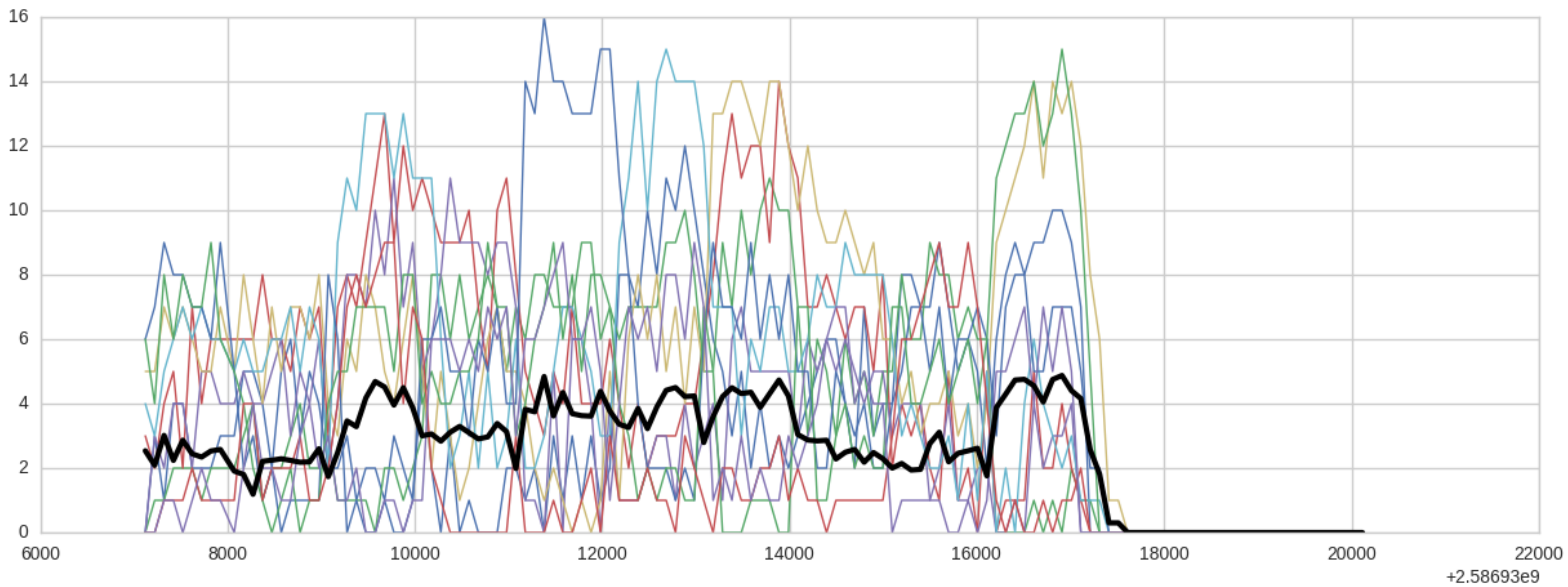
LET'S THROW A WRENCH INTO THIS...

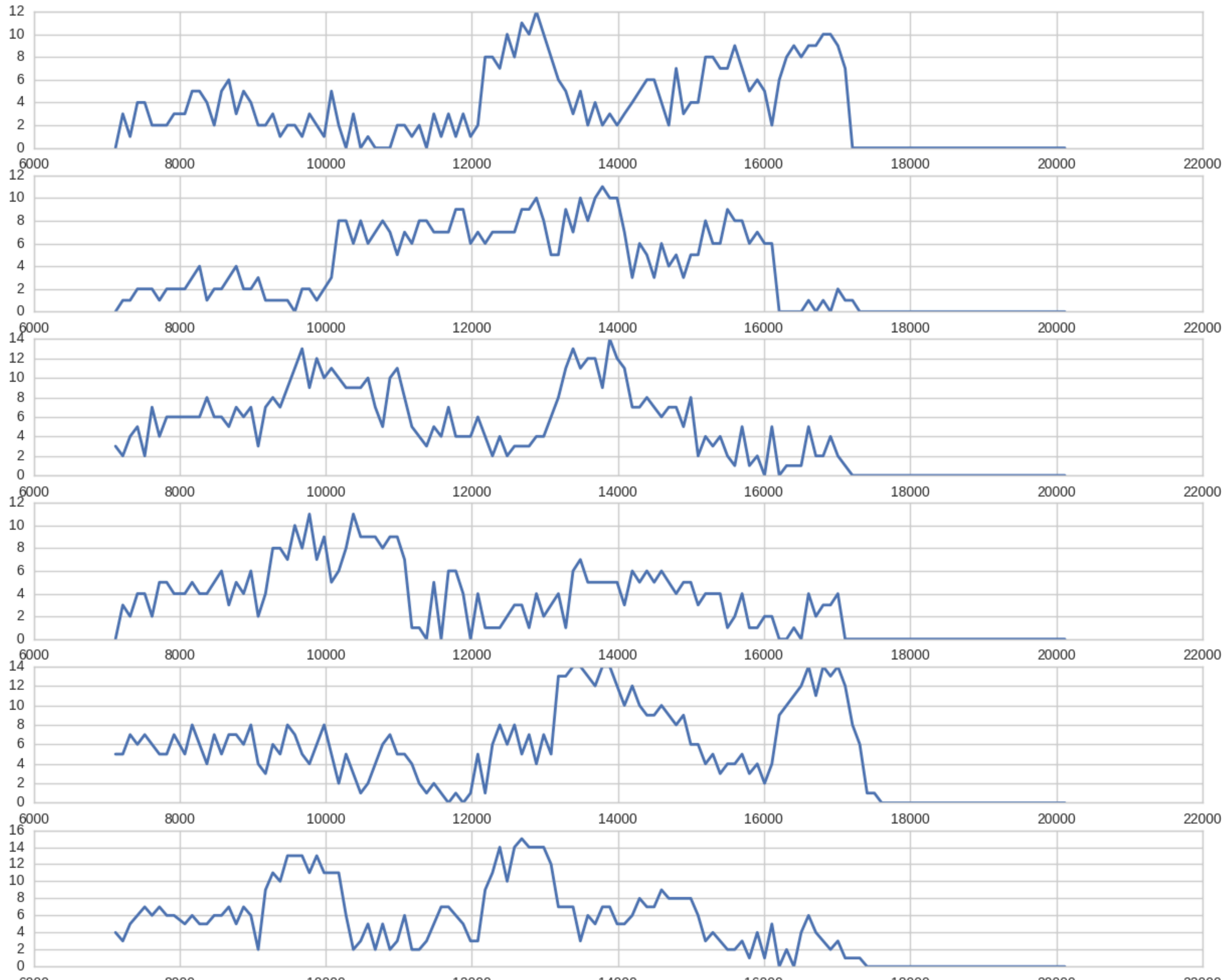
DISTRIBUTED LOAD BALANCING

AND WHY IT MAKES EVERYTHING HARDER

DISTRIBUTED RANDOM IS EXACTLY THE SAME

DISTRIBUTED JOIN-SHORTEST-QUEUE IS A NIGHTMARE


16
14
12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
+2.58693e9



12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
14
12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
14
12
10
8
6
4
2
0
6000
8000
10000
12000
14000
16000
18000
20000
22000
16
14
12
10
8
6
4
2
0

The Power of Two Choices in Randomized Load Balancing

by

Michael David Mitzenmacher

B.A. (Harvard University) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the

```python
mu = 0.0
sigma = 1.15
lognorm_mean = math.e ** (mu + sigma ** 2 / 2)

desired_mean = 1.0
baseline = 0.05

def normalize(value):
    return (value / lognorm_mean
            * (desired_mean - baseline)
            + baseline)

for weight in nr.lognormal(mu, sigma, m):
    chosen_bin = nr.randint(0, n)
    bins[chosen_bin] += normalize(weight)
```

[100.7, 137.5, 134.3, 126.2, 113.5, 175.7, 101.6, 113.7]

```python
mu = 0.0
sigma = 1.15
lognorm_mean = math.e ** (mu + sigma ** 2 / 2)

desired_mean = 1.0
baseline = 0.05

def normalize(value):
    return (value / lognorm_mean
            * (desired_mean - baseline)
            + baseline)

for weight in nr.lognormal(mu, sigma, m):
    a = nr.randint(0, n)
    b = nr.randint(0, n)
    chosen_bin = a if bins[a] < bins[b] else b
    bins[chosen_bin] += normalize(weight)
```

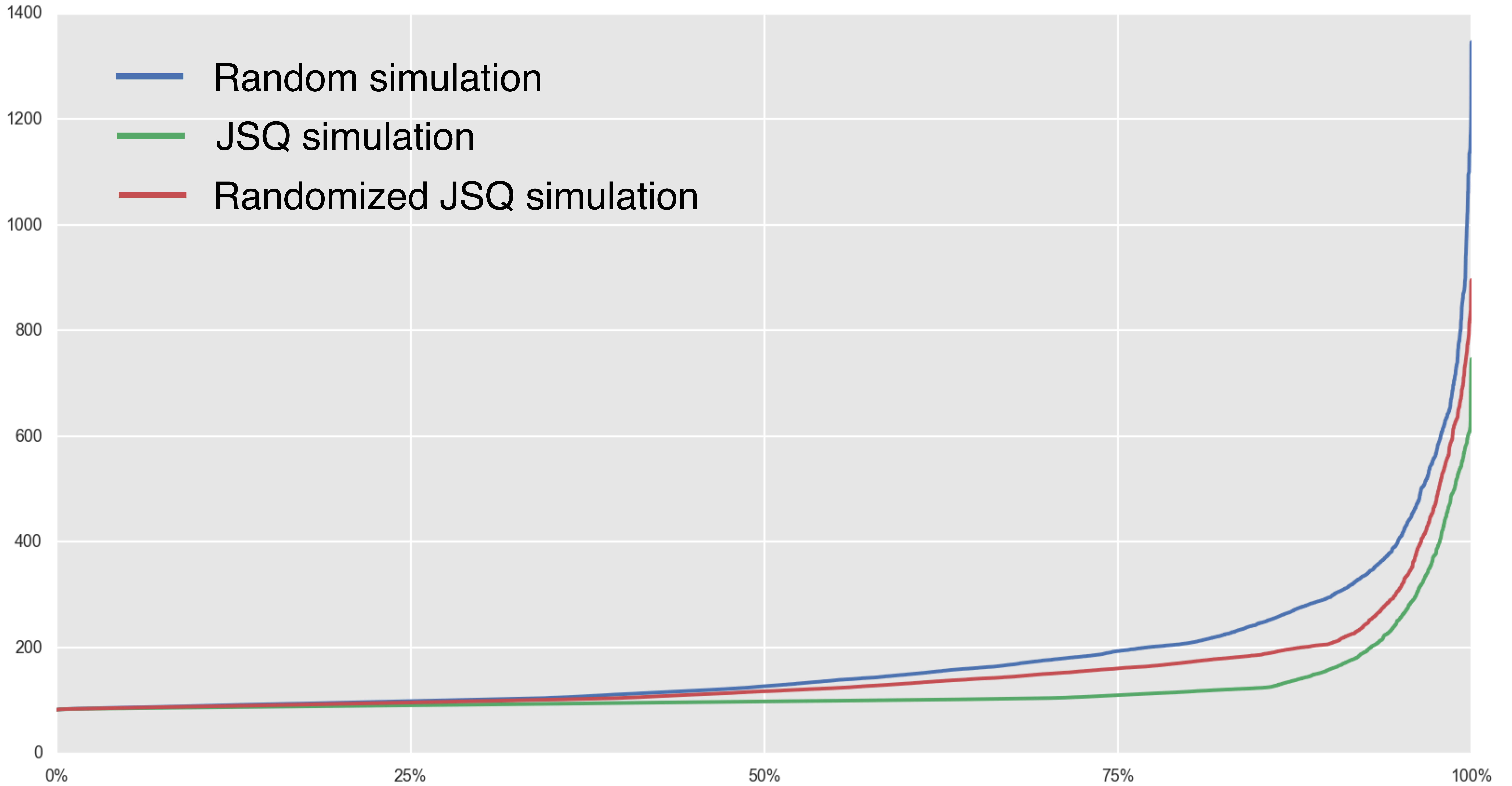
[130.5, 131.7, 129.7, 132.0, 131.3, 133.2, 129.9, 132.6]

[100.7, 137.5, 134.3, 126.2, 113.5, 175.7, 101.6, 113.7]

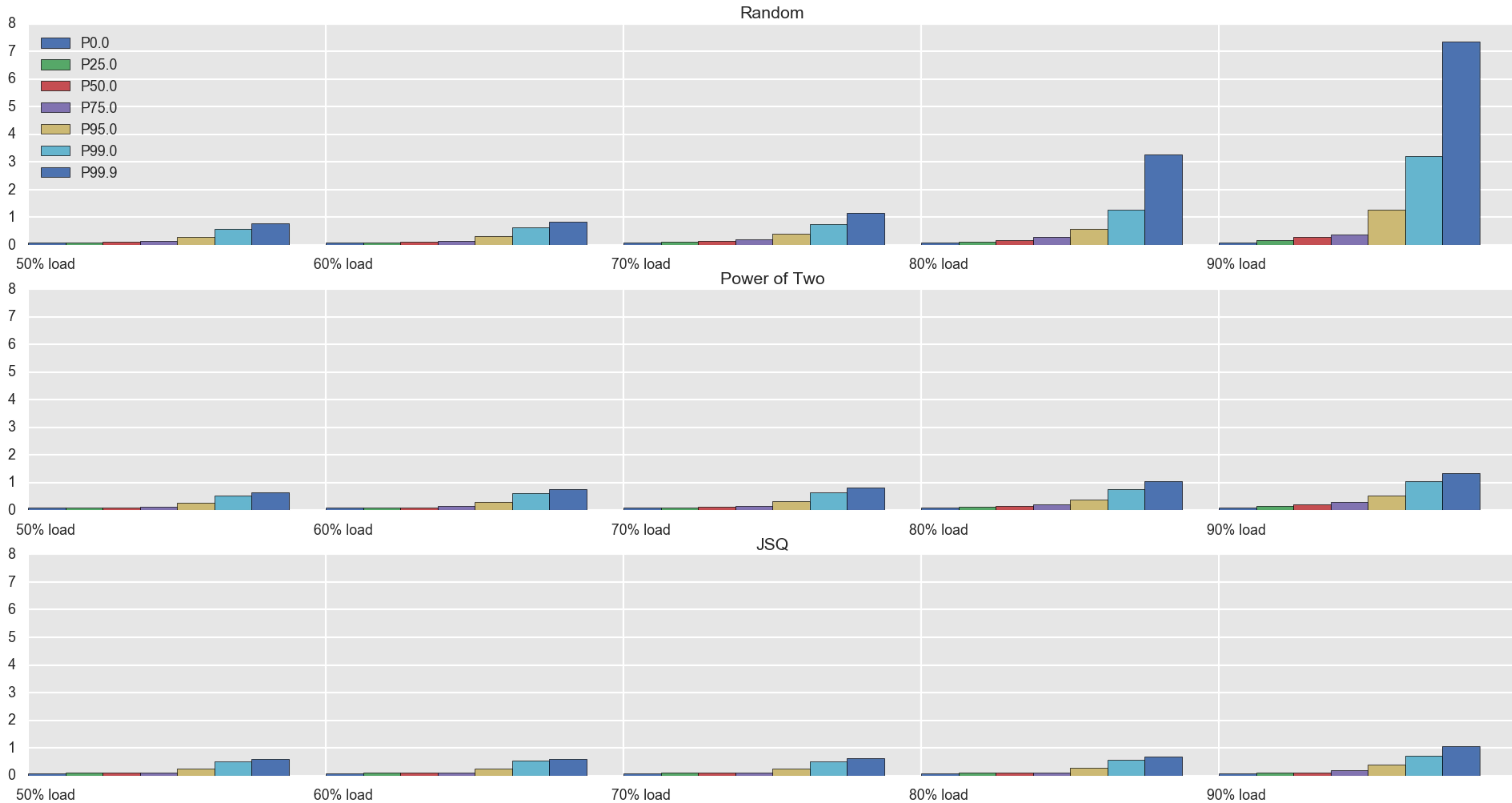
STANDARD DEVIATION: 22.9

[130.5, 131.7, 129.7, 132.0, 131.3, 133.2, 129.9, 132.6]

STANDARD DEVIATION: 1.18



Random simulation
JSQ simulation
Randomized JSQ simulation
1400
1200
1000
800
600
400
200
0
0%
25%
50%
75%
100%


Random
P0.0
P25.0
P50.0
P75.0
P95.0
P99.0
P99.9
0
1
2
3
4
5
6
7
8
50% load
60% load
70% load
80% load
90% load
Power of Two
0
1
2
3
4
5
6
7
8
50% load
60% load
70% load
80% load
90% load
JSQ
0
1
2
3
4
5
6
7
8
50% load
60% load
70% load
80% load
90% load

How Useful is Old Information?

Michael Mitzenmacher
Digital Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301
michaelm@pa.dec.com

Abstract

We consider the problem of load balancing in dynamic distributed systems in cases where new incoming tasks can make use of old information. For example, consider a multi-processor system where incoming tasks with exponentially distributed service requirements arrive as a Poisson process, the tasks must choose a processor for service, and a task knows when making this choice the processor loads from T seconds ago. What is a good strategy for choosing a processor, in order for tasks to minimize their expected time in the system? Such models can also be used to describe settings where there is a transfer delay between the time a task enters a system and the time it reaches a processor for service.

Our models are based on considering the behavior of limiting systems where the number of processors goes to infinity. The limiting systems can be shown

Interpreting Stale Load Information*

Michael Dahlin
Department of Computer Sciences
University of Texas at Austin
dahlin@cs.utexas.edu

Abstract

In this paper we examine the problem of balancing load in a large-scale distributed system when information about server loads may be stale. It is well known that sending each request to the machine with the apparent lowest load can behave badly in such systems, yet this technique is common in practice. Other systems use round-robin or random selection algorithms that entirely ignore load information or that only use a small subset of the load information. Rather than risk extremely bad performance on one hand or ignore the chance to use load information to improve performance on the other, we develop strategies that interpret load information based on its age. Through simulation, we examine several simple algorithms that use such load interpretation strategies under a range of workloads. Our experiments suggest that by properly interpreting load information, systems can (1) match the performance of the most aggressive algorithms when load information is fresh relative to the job arrival rate, (2) outperform the best of the other algorithms we examine by as much as 60% when information is moderately old, (3) significantly outperform random load distribution when information is older still, and (4) avoid pathological behavior even when information is extremely old.

C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection

Lalith Suresh[†] Marco Canini[⋆] Stefan Schmid[†‡] Anja Feldmann[†]
[†]*TU Berlin* [⋆]*Université catholique de Louvain* [‡]*Telekom Innovation Labs*

Abstract

Achieving predictable performance is critical for many distributed applications, yet difficult to achieve due to many factors that skew the tail of the latency distribution even in well-provisioned systems. In this paper, we present the fundamental challenges involved in designing a replica selection scheme that is robust in the face of performance fluctuations across servers. We illustrate these challenges through performance evaluations of the Cassandra distributed database on Amazon EC2. We then present the design and implementation of an adaptive replica selection mechanism, C3, that is robust to performance variability in the environment. We demonstrate C3's effectiveness in reducing the latency tail and improving throughput through extensive evaluations on Amazon EC2 and through simulations. Our results show that C3 significantly improves the latencies along the mean, median, and tail (up to 3 times improvement at the

cial clouds to deliver applications further exacerbates the response time unpredictability since, in these environments, applications almost unavoidably experience performance interference due to contention for shared resources (like CPU, memory, and I/O) [26, 50, 52].

Several studies [16, 23, 50] indicate that latency distributions in Internet-scale systems exhibit long-tail behaviors. That is, the 99.9^{th} percentile latency can be more than an order of magnitude higher than the median latency. Recent efforts [2, 16, 19, 23, 36, 44, 53] have thus proposed approaches to reduce tail latencies and lower the impact of skewed performance. These approaches rely on standard techniques including giving preferential resource allocations or guarantees, reissuing requests, trading off completeness for latency, and creating performance models to predict stragglers in the system.

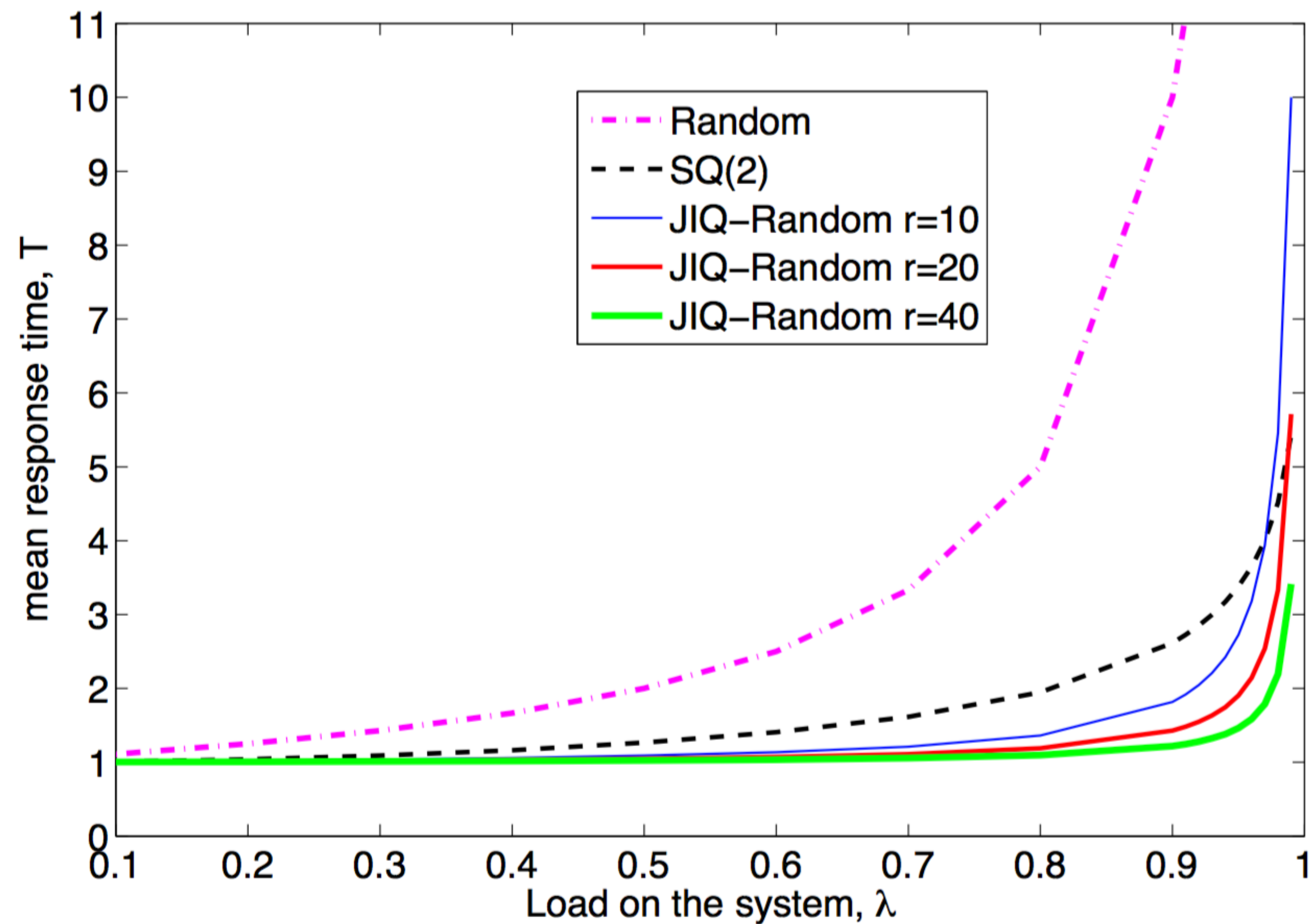
A recurring pattern to reducing tail latency is to ex-

ANOTHER CRAZY IDEA

Join-Idle-Queu
for Dyn

Yi Lu[a], Qiaomin Xie[a], Gabri

[a]Department of Electrical and Co
[b]Extreme Computing Group, Mic
[c]Microsoft Azure



(a)

The prevalence of dynamic-
has motivated an increasingly wi
elasticity, and distributed design
ized design, such as Join-the-Sh
dispatchers.

We propose a novel class of
in large systems. Unlike algorit
overhead between the dispatche
large system limit and find that
reduction in queueing overhead
basic JIQ algorithm deals with very high loads using only local information of server load.

Keywords: Load balancing · queueing analysis · randomized algorithm · cloud computing

1 Introduction

WRAP UP

THANKS BYE

tyler@fastly.com

@tbmcmullen