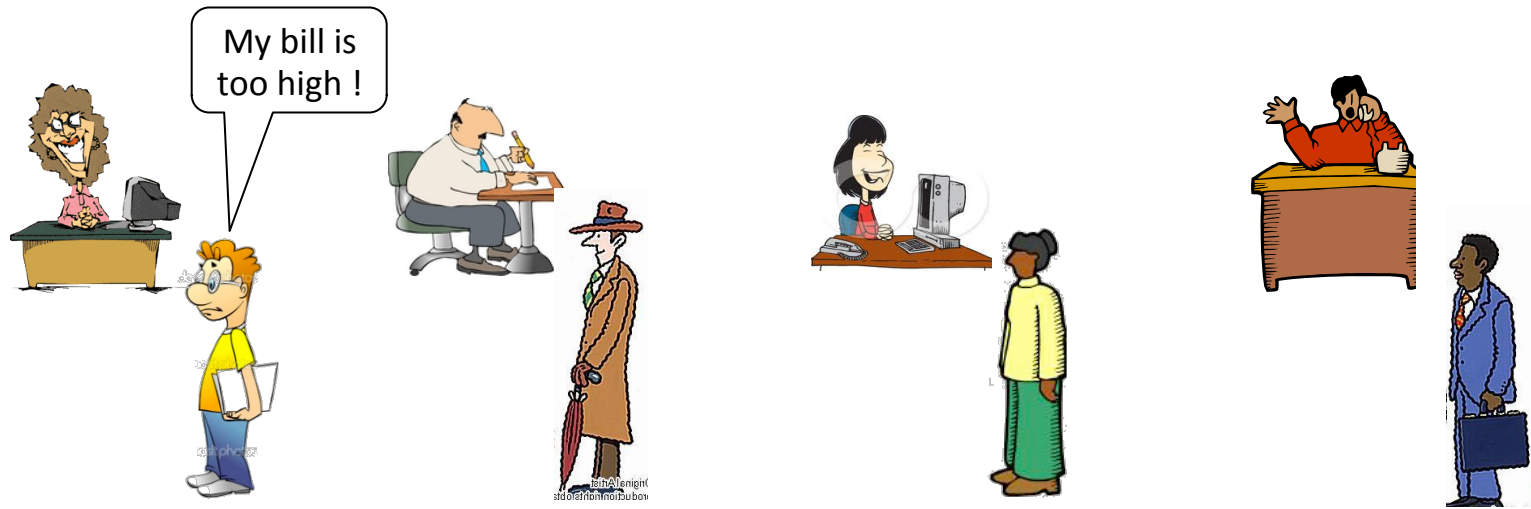
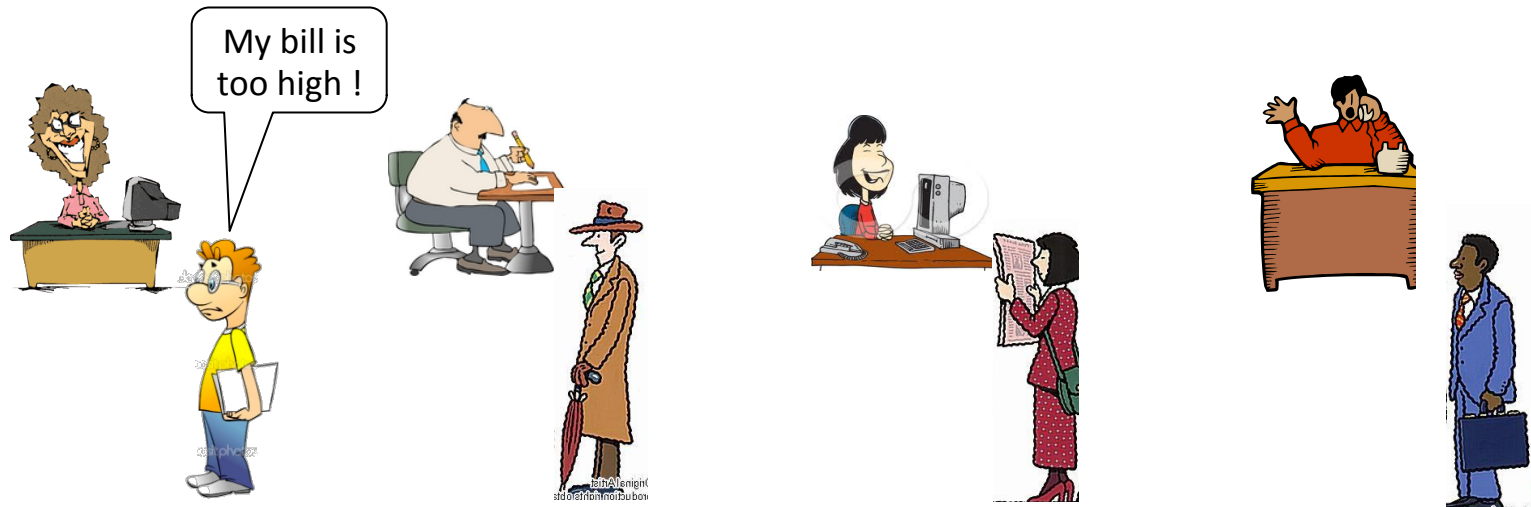


When Do Redundant Requests Reduce Latency ?

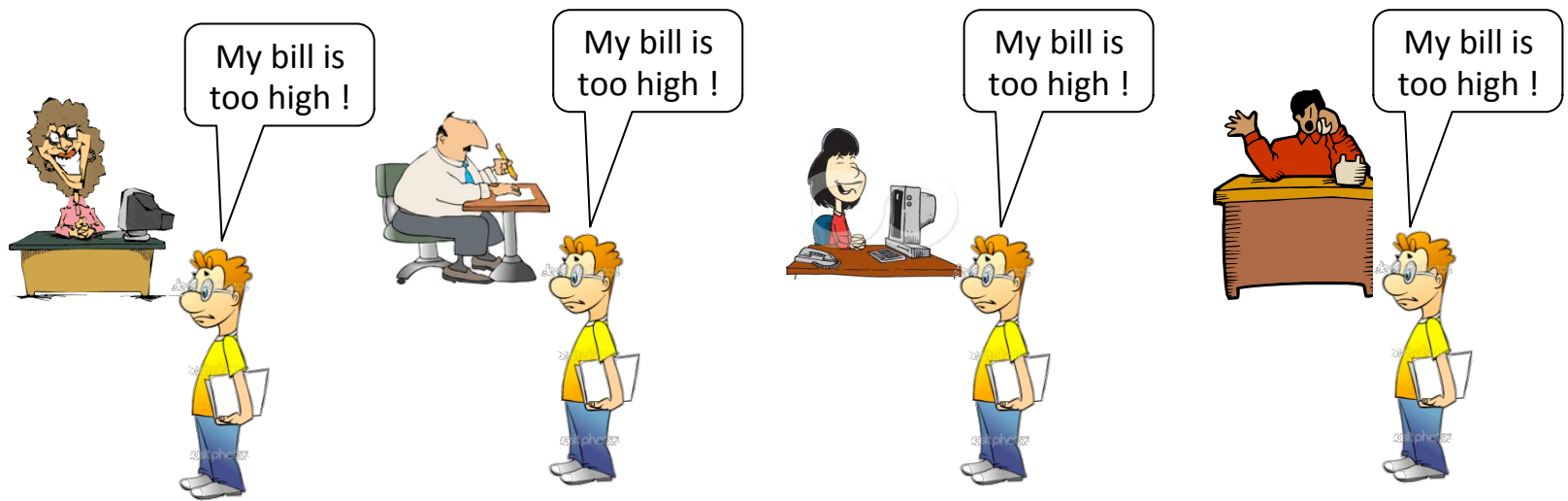
Nihar B. Shah,
Kangwook Lee, Kannan Ramchandran
UC Berkeley

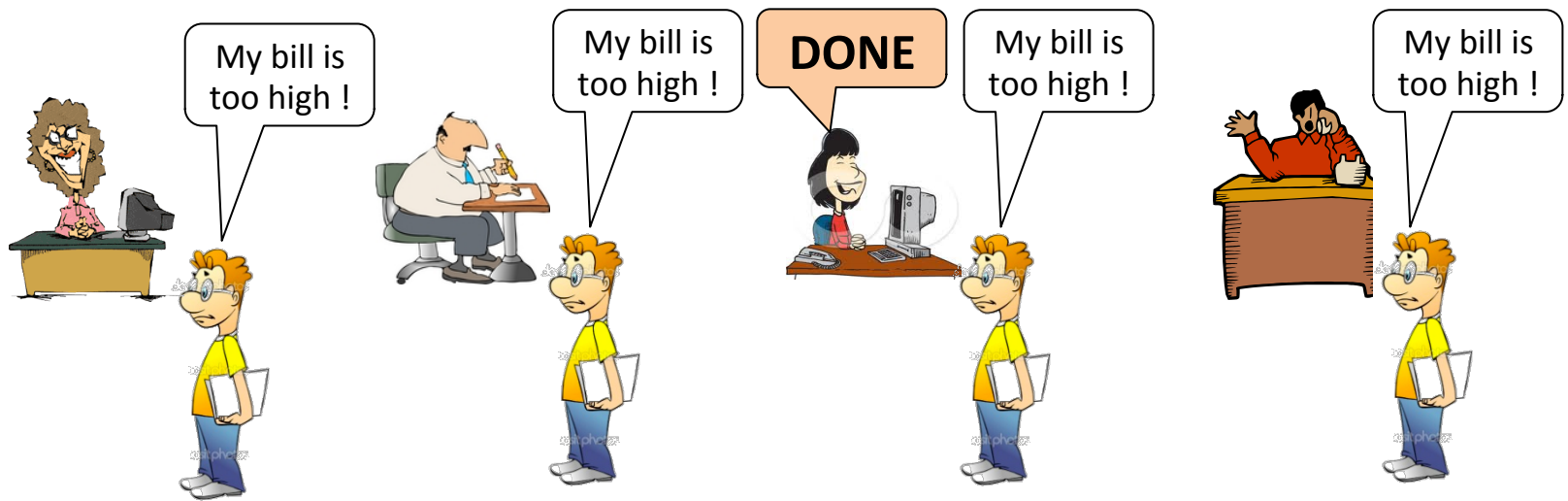






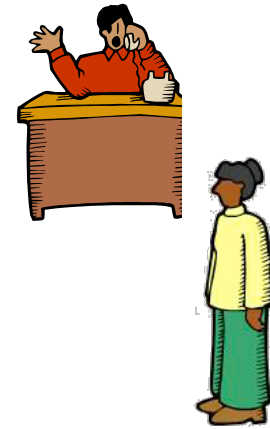
Alternatively...







Or

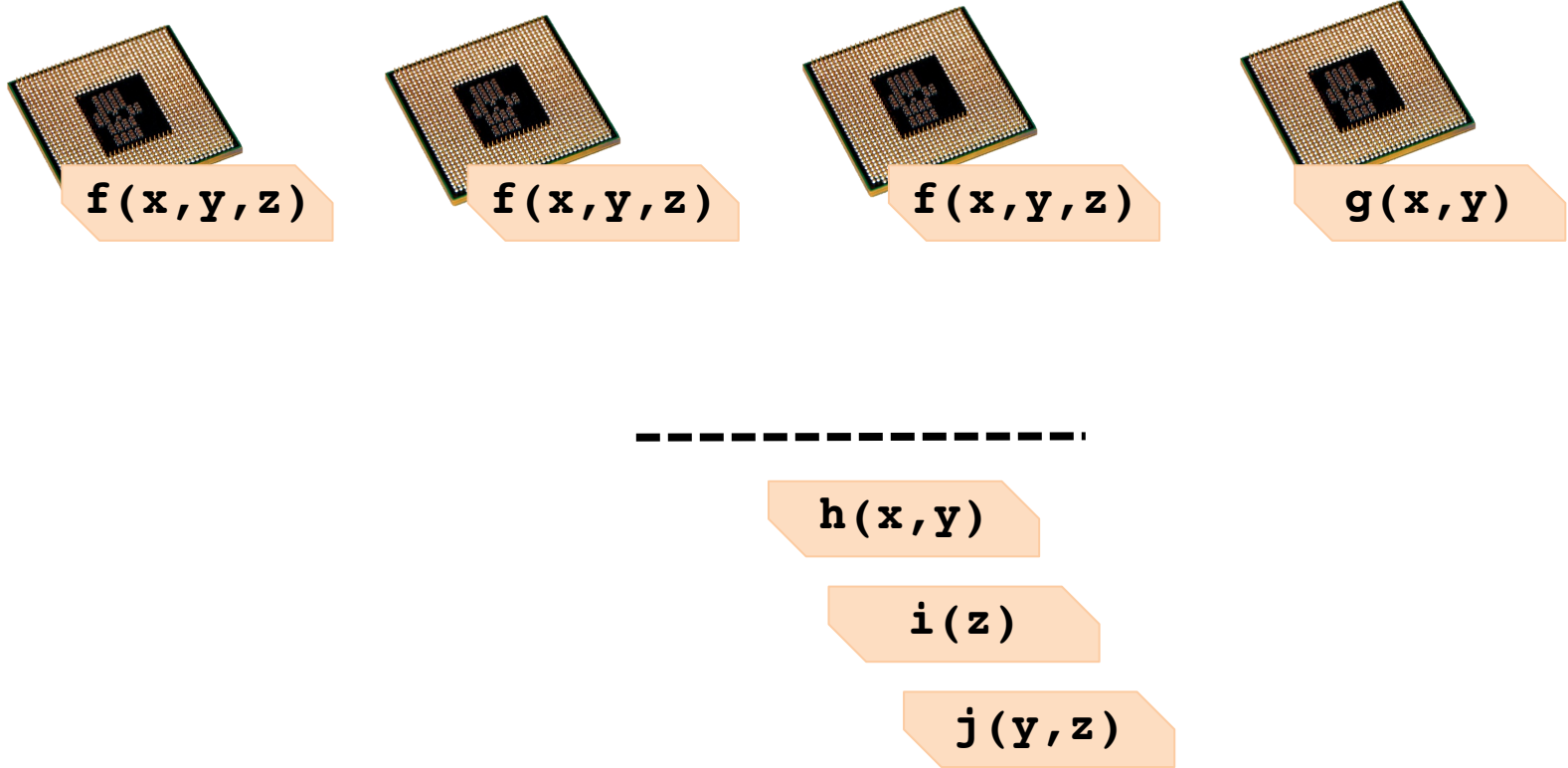




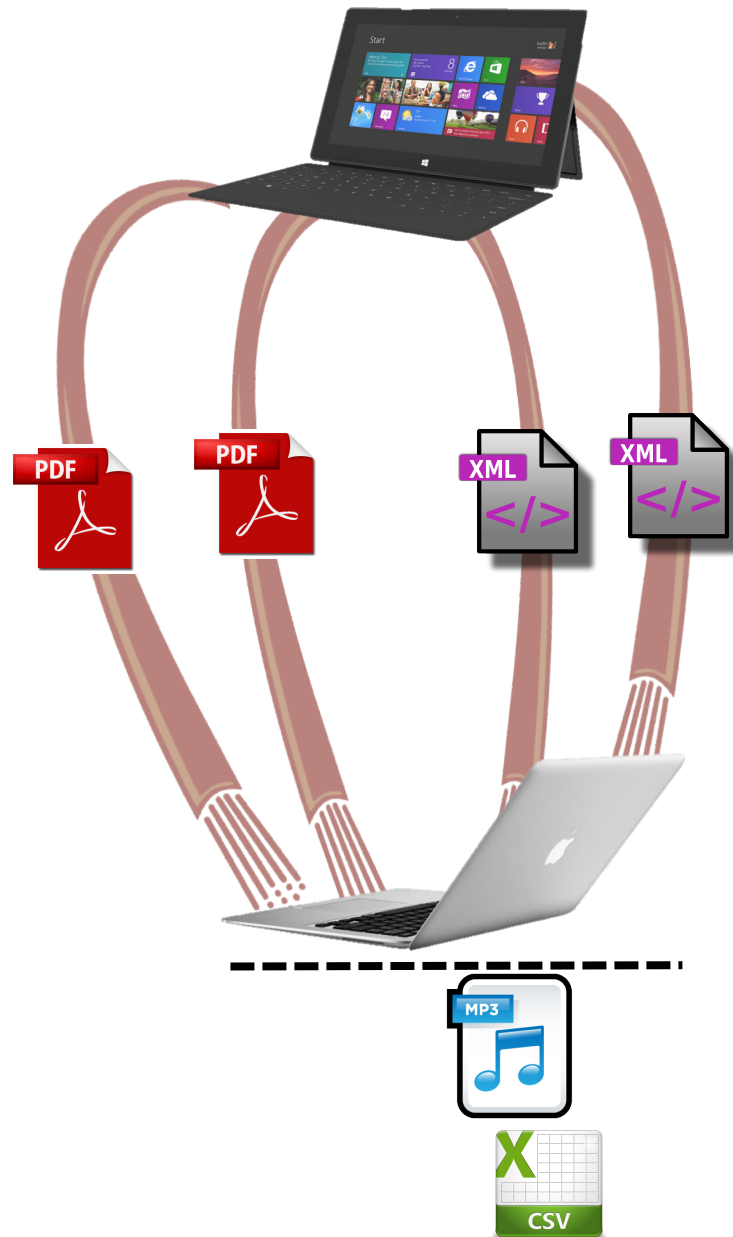
Redundant Requests



processors in compute cluster



multiple transmission paths



storage system that stores data redundantly
e.g., (4, 2) Reed-Solomon code



\$ get file G



\$ get file G



\$ get file G



\$ get file F

\$ get file F

\$ get file H

Redundant Requests: Tradeoff

- ✓ More servers process each request
 - ✓ Processing time reduces
- × More resources consumed
 - × Increase in queuing delay

When do they reduce latency?

DOI:10.1145/2408776.2408794

Software techniques that tolerate latency variability are vital to building responsive large-scale Web services.

BY JEFFREY DEAN AND LUIZ ANDRÉ BARROSO

The Tail at Scale

SYSTEMS THAT RESPOND to user actions quickly (within 100ms) feel more fluid and natural to users than those that take longer.¹ Improvements in Internet connectivity and the rise of warehouse-scale computing systems² have enabled Web services that provide fluid responsiveness while consulting multi-terabyte datasets spanning thousands of servers; for example, the Google search system updates query results interactively as the user types, predicting the most likely query based on the prefix typed so far, performing the search and showing the results within a few tens of milliseconds. Emerging augmented-reality devices (such as the Google Glass prototype³) will need associated Web services with even greater responsiveness in order to guarantee seamless interactivity.

It is challenging for service providers to keep the tail of latency distribution short for interactive services as the size and complexity of the system scales up or

as overall use increases. Temporary high-latency episodes (unimportant in moderate-size systems) may come to dominate overall service performance at large scale. Just as fault-tolerant computing aims to create a reliable whole out of less-reliable parts, large online services need to create a predictably responsive whole out of less-predictable parts; we refer to such systems as “latency tail-tolerant,” or simply “tail-tolerant.” Here, we outline some common causes for high-latency episodes in large online services and describe techniques that reduce their severity or mitigate their effect on whole-system performance. In many cases, tail-tolerant techniques can take advantage of resources already deployed to achieve fault-tolerance, resulting in low additional overhead. We explore how these techniques allow system utilization to be driven higher without lengthening the latency tail, thus avoiding wasteful overprovisioning.

Why Variability Exists?

Variability of response time that leads to high tail latency in individual components of a service can arise for many reasons, including:

Shared resources. Machines might be shared by different applications contending for shared resources (such as CPU cores, processor caches, memory bandwidth, and network bandwidth), and within the same application different requests might contend for resources;

Daemons. Background daemons may use only limited resources on average but when scheduled can generate multi-millisecond hiccups;

» key insights

- Even rare performance hiccups affect a significant fraction of all requests in large-scale distributed systems.
- Eliminating all sources of latency variability in large-scale systems is impractical, especially in shared environments.
- Using an approach analogous to fault-tolerant computing, tail-tolerant software techniques form a predictable whole out of less-predictable parts.

ILLUSTRATION BY ELLIOTT HANCOCK

Erasure Coding in Windows Azure Storage

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin

Microsoft Corporation

Abstract

Windows Azure Storage (WAS) is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data for any duration of time. WAS customers have access to their data from anywhere, at any time, and only pay for what they use and store. To provide durability for that data and to keep the cost of storage low, WAS uses erasure coding.

In this paper we introduce a new set of codes for erasure coding called Local Reconstruction Codes (LRC). LRC reduces the number of erasure coding fragments that need to be read when reconstructing data fragments that are offline, while still keeping the storage overhead low. The important benefits of LRC are that it reduces the bandwidth and I/Os required for repair reads over prior codes, while still allowing a significant reduction in storage overhead. We describe how LRC is used in WAS to provide low overhead durable storage with consistently low read latencies.

1 Introduction

Windows Azure Storage (WAS) [1] is a scalable cloud storage system that has been in production since November 2008. It is used inside Microsoft for applications such as social networking search, serving video, music and game content, managing medical records, and more. In addition, there are thousands of customers outside Microsoft using WAS, and anyone can sign up over the Internet to use the system. WAS provides cloud storage in the form of Blobs (user files), Tables (structured storage), Queues (message delivery), and Drives (network mounted VHDs). These data abstractions provide the overall storage and work flow for applications running in the cloud.

WAS stores all of its data into an append-only distributed file system called the stream layer [1]. Data is appended to the end of active *extents*, which are replicated three times by the underlying stream layer. The data is originally written to 3 full copies to keep the data durable. Once reaching a certain size (e.g., 1 GB), extents are sealed. These sealed extents can no longer be modified and thus make perfect candidates for erasure coding. WAS then erasure codes a sealed extent lazily in the background, and once the extent is erasure-coded the original 3 full copies of the extent are deleted.

The motivation for using erasure coding in WAS comes from the need to reduce the cost of storage. Erasure coding can reduce the cost of storage over 50%,

which is a tremendous cost saving as we will soon surpass an Exabyte of storage. There are the obvious cost savings from purchasing less hardware to store that much data, but there are significant savings from the fact that this also reduces our data center footprint by 1/2, the power savings from running 1/2 the hardware, along with other savings.

The trade-off for using erasure coding instead of keeping 3 full copies is performance. The performance hit comes when dealing with *i)* a lost or offline data fragment and *ii)* hot storage nodes. When an extent is erasure-coded, it is broken up into *k*-data fragments, and a set of parity fragments. In WAS, a data fragment may be lost due to a disk, node or rack failure. In addition, cloud services are *perpetually in beta* [2] due to frequent upgrades. A data fragment may be offline for seconds to a few minutes due to an upgrade where the storage node process may be restarted or the OS for the storage node may be rebooted. During this time, if there is an on-demand read from a client to a fragment on the storage node being upgraded, WAS reads from enough fragments in order to dynamically reconstruct the data being asked for to return the data to the client. This reconstruction needs to be optimized to be as fast as possible and use as little networking bandwidth and I/Os as possible, with the goal to have the reconstruction time consistently low to meet customer SLAs.

When using erasure coding, the data fragment the client's request is asking for is stored on a specific storage node, which can greatly increase the risk of a storage node becoming hot, which could affect latency. One way that WAS can deal with hot storage nodes is to recognize the fragments that are hot and then replicate them to cooler storage nodes to balance out the load, or cache the data and serve it directly from DRAM or SSDs. But, the read performance can suffer for the potential set of reads going to that storage node as it gets hot, until the data is cached or load balanced. Therefore, one optimization WAS has is if it looks like the read to a data fragment is going to take too long, WAS in parallel tries to perform a reconstruction of the data fragment (effectively treating the storage node with the original data fragment as if it was offline) and return to the client whichever of the two results is faster.

For both of the above cases the time to reconstruct a data fragment for on-demand client requests is crucial. The problem is that the reconstruction operation is only as fast as the slowest storage node to respond to reading

Why let resources idle? Aggressive Cloning of Jobs with Dolly

Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, Ion Stoica

University of California, Berkeley

{ganesha, alig, shenker, istoica}@cs.berkeley.edu

Abstract

Despite prior research on outlier mitigation, our analysis of jobs from the Facebook cluster shows that outliers still occur, especially in small jobs. Small jobs are particularly sensitive to long-running outlier tasks because of their interactive nature. Outlier mitigation strategies rely on comparing different tasks of the same job and launching speculative copies for the slower tasks. However, small jobs execute all their tasks simultaneously, thereby not providing sufficient time to observe and compare tasks. Building on the observation that clusters are underutilized, we take speculation to its logical extreme—*run full clones of jobs to mitigate the effect of outliers*. The heavy-tail distribution of job sizes implies that we can impact most jobs without using much resources. Trace-driven simulations show that average completion time of all the small jobs improves by 47% using cloning, at the cost of just 3% extra resources.

1 Introduction

Cloud computing has become a significant technological breakthrough. An increasing number of organizations use datacenters to run a mixed variety of computations, ranging from long-running batch jobs to interactive short queries that operators launch on the fly.

The importance of these datacenter computations has led to much effort being spent on optimizing their performance. The prevalence of outlier tasks was early identified as a common source of performance problem [1]. Initial research suggested the use of speculative execution to mitigate such outliers. These methods were later improved by LATE [2] and Mantri [3], which provide more intelligent outlier mitigation based on speculative execution of tasks. Similar techniques have also been used to deal with outliers in other settings [4, 5].

Despite this research on outlier mitigation, our analyses of traces from a 3,500 node Facebook cluster, that applies the LATE technique, shows that outliers are still common, especially in small jobs. The small jobs, on av-

erage, have outlier tasks that are 12 times slower than that job's median task, which significantly delays completion of jobs. Our simulations show that the outlier numbers for Mantri are similar for small jobs.

Small jobs are particularly sensitive to outliers because they execute in a single wave of simultaneously running tasks. Therefore even a single task being an outlier slows down the entire job. The single-waved property also limits the efficacy of traditional outlier mitigation strategies that rely on comparing different tasks of the same job. Any meaningful comparison requires waiting to obtain statistically significant samples of task performance, which single-waved small jobs cannot afford.

In this work, we focus on improving the completion time of these small jobs, which are often interactive queries, where the response time is important to the human operator awaiting its response. The idea we explore in this paper is to take speculative execution to its logical extreme and *run full clones of jobs to reduce job completion times*. Two trends make this approach viable.

First, most jobs are small and consume few resources. Our analysis shows that job sizes have a power-law distribution, with the absolute majority of the jobs being small, while the absolute majority of the cluster resources are spent on a small number of large jobs. Thus, the aggregate resources consumed by small jobs is moderate. Running clones of small jobs has the potential to impact most jobs, without using much resources.

Second, most clusters are highly underutilized. Several of the clusters that we analyzed have a very low average utilization. In particular, CPU and memory utilization in these clusters has a median less than 20%. In fact, cluster utilization exceeds the 50% mark only 8% of the time. There is thus room for running extra clones of jobs.

A key question is whether running job clones will negatively impact energy efficiency. Despite research on powering down machines for energy efficiency, we note that most clusters today *do not* shut off machines to save energy. Thus, machines are on most of the

Dean & Barroso '13, Huang et al. '12, Ananthanarayanan et al. '12

More is Less: Reducing Latency via Redundancy

Ashish Vulimiri
UIUC
vulimiri1@illinois.edu

Oliver Michel
University of Vienna
oliver.michel@univie.ac.at

P. Brighten Godfrey
UIUC
pbg@illinois.edu

Scott Shenker
UC Berkeley and ICSI
shenker@icsi.berkeley.edu

ABSTRACT

Low latency is critical for interactive networked applications. But while we know how to scale systems to increase capacity, reducing latency — especially the tail of the latency distribution — can be much more difficult.

We argue that the use of redundancy in the context of the wide-area Internet is an effective way to convert a small amount of extra capacity into reduced latency. By initiating redundant operations across diverse resources and using the first result which completes, redundancy improves a system's latency even under exceptional conditions. We demonstrate that redundancy can significantly reduce latency for small but critical tasks, and argue that it is an effective general-purpose strategy even on devices like cell phones where bandwidth is relatively constrained.

Categories and Subject Descriptors

C.2.0 [Computer-Communication Networks]: General

General Terms

Performance, Reliability

1. INTRODUCTION

Low latency is important for humans. Even slightly higher web page load times can significantly reduce visits from users and revenue, as demonstrated by several sites [21]. For example, injecting just 400 milliseconds of artificial delay into Google search results caused the delayed users to perform 0.74% fewer searches after 4-6

weeks [7]. A 500 millisecond delay in the Bing search engine reduced revenue per user by 1.2%, or 4.3% with a 2-second delay [21]. Human-computer interaction studies similarly show that people react to small differences in the delay of operations (see [12] and references therein).

Achieving consistent low latency is challenging. Modern applications are highly distributed, and likely to get more so as cloud computing separates users from their data and computation. Moreover, application-level operations often require tens or hundreds of tasks to complete — due to many objects comprising a single web page [19], or aggregation of many back-end queries to produce a front-end result [1,10]. This means individual tasks may have latency budgets on the order of a few milliseconds or tens of milliseconds, and *the tail* of the latency distribution is critical. Thus, latency is a difficult challenge for networked systems: How do we make the other side of the world feel like it is *right here*, even under exceptional conditions?

One powerful technique to reduce latency is *redundancy*: Initiate an operation multiple times, using as diverse resources as possible, and use the first result which completes. For example, a host may query multiple DNS servers in parallel to resolve a name. The overall latency is the minimum of the delays across each instance, thus potentially reducing both the mean and the tail of the latency distribution. The power of this technique is that it *reduces latency precisely under the most challenging conditions*: when delays or failures are unpredictable.

Redundancy has been employed in several past networked systems: notably, as a way to deal with failures in DTNs [15], and in a multi-homed web proxy overlay [3]. But beyond these specific research projects, redundancy is typically eschewed across the Internet. We argue this is a missed opportunity.

The contribution of this paper is to argue for redundancy as a general technique for the wide-area Internet. The combination of interactive applications, high latency, and variability of latency make redundancy well suited to this environment. Even in a well-provisioned

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Hotnets '12, October 29–30, 2012, Seattle, WA, USA.
Copyright 2012 ACM 978-1-4503-1776-4/10/12 ...\$10.00.

Dean & Barroso '13, Huang et al. '12, Ananthanarayanan et al. '12, Vulimiri et al. '12

FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding

Guanfeng Liang, *Member, IEEE*, and Ulaş C. Kozat, *Senior Member, IEEE*

Abstract—Our paper presents solutions that can significantly improve the delay performance of putting and retrieving data in and out of cloud storage. We first focus on measuring the delay performance of a very popular cloud storage service Amazon S3. We establish that there is significant randomness in service times for reading and writing small and medium size objects when assigned distinct keys. We further demonstrate that using erasure coding, parallel connections to storage cloud and limited chunking (i.e., dividing the object into a few smaller objects) together pushes the envelope on service time distributions significantly (e.g., 76%, 80%, and 85% reductions in mean, 90th, and 99th percentiles for 2 Mbyte files) at the expense of additional storage (e.g., 1.75×). However, chunking and erasure coding increase the load and hence the queuing delays while reducing the supportable rate region in number of requests per second per node. Thus, in the second part of our paper we focus on analyzing the delay performance when chunking, FEC, and parallel connections are used together. Based on this analysis, we develop load adaptive algorithms that can pick the best code rate on a per request basis by using off-line computed queue backlog thresholds. The solutions work with homogeneous services with fixed object sizes, chunk sizes, operation type (e.g., read or write) as well as heterogeneous services with mixture of object sizes, chunk sizes, and operation types. We also present a simple greedy solution that opportunistically uses idle connections and picks the erasure coding rate accordingly on the fly. Both backlog and greedy solutions support the full rate region and provide best mean delay performance when compared to the best fixed coding rate policy. Our evaluations show that backlog based solutions achieve better delay performance at higher percentile values than the greedy solution.

Index Terms—FEC, Cloud storage, Queueing, Delay

I. INTRODUCTION

Public clouds have been utilized by web services and Internet applications widespread. They provide high degree of availability, scalability, and data durability. Yet, there exists significant skew in network bound I/O performance necessitating solutions that provide robustness in a cost effective manner [1], [2]. In this paper, we focus on the cloud storage and present solutions that can provide much better delay performance for putting files into the cloud storage as well as for retrieving them back on demand. In particular, we base our analysis on Amazon S3 service as one of the most popular cloud storage service.

A typical cloud storage stores and retrieves objects via their unique keys. Each object is replicated several times within the cloud and sometimes also further protected by erasure codes to more efficiently use the storage capacity while attaining very

high durability guarantees [3]. Storage provider also monitors the load on each storage node and employs dynamic load balancing to prevent hot storage nodes that might observe high loads or slow nodes that have excessively high response times. Although mainly used for repairing data in unavailable storage nodes, some cloud providers also access coded blocks in parallel to uncoded blocks when uncoded blocks are stored in slow nodes [3]. Despite all these mechanisms, still evaluations of large scale systems indicate that there is a high degree of randomness in delay performance [1]. Thus, the services that require better delay performance must deploy their own solutions such as sending multiple requests (in parallel or sequentially), chunking large objects into smaller ones and read/write each chunk in parallel, replicate the same object using multiple distinct keys, etc.

To this end, we conducted our own measurements on Amazon S3 for various object sizes to model its delay distribution. Our measurement results confirm that the delay spread is significant even when object sizes are in the order of megabytes. Moreover, our study indicates that when the server accessing the storage cloud is not the bottleneck (in terms of CPU and network access speed), we can substantially improve the distribution of read/write delays. To achieve these gains, one has to consider not only chunking and parallel access to each chunk, but also erasure coding. In fact without erasure coding, more chunking starts hurting the performance at lower percentile values. The gains when forward error correction (FEC) is employed are significant in the average delay performance and they are much better at higher percentile delays.

Nonetheless, server accessing the storage cloud has limited CPU and network access speed limiting the number of concurrent connections to the storage cloud without going into a processor sharing mode. With limited system capacity, one has to consider the load and its impact on queueing delays to quantify the total delay. Unfortunately, FEC and chunking create redundant load multiplying the arrival rate into the system. Unless mean service rate is improved to the same extent, the maximum rate at which end users can be served is reduced. Our observations over Amazon S3 indicate that indeed lower code rates reduce the supportable rate region inducing queue instability earlier than higher code rates. Thus, it is imperative to design a load adaptive strategy for changing FEC rates on the fly to keep total average delays at the minimum level while remaining in the achievable rate region of the uncoded system.

To come up with meaningful solutions, one needs to analyze queueing delay for the system. As one of the main contributions of the paper, we analyze the average delay performance of a

arXiv:1301.1294v1 [cs.NI] 7 Jan 2013

G. Liang and U.C. Kozat are with DOCOMO Innovations Inc., Palo Alto, California USA. G. Liang is the contact author. E-mail: giliang@docomoinnovations.com

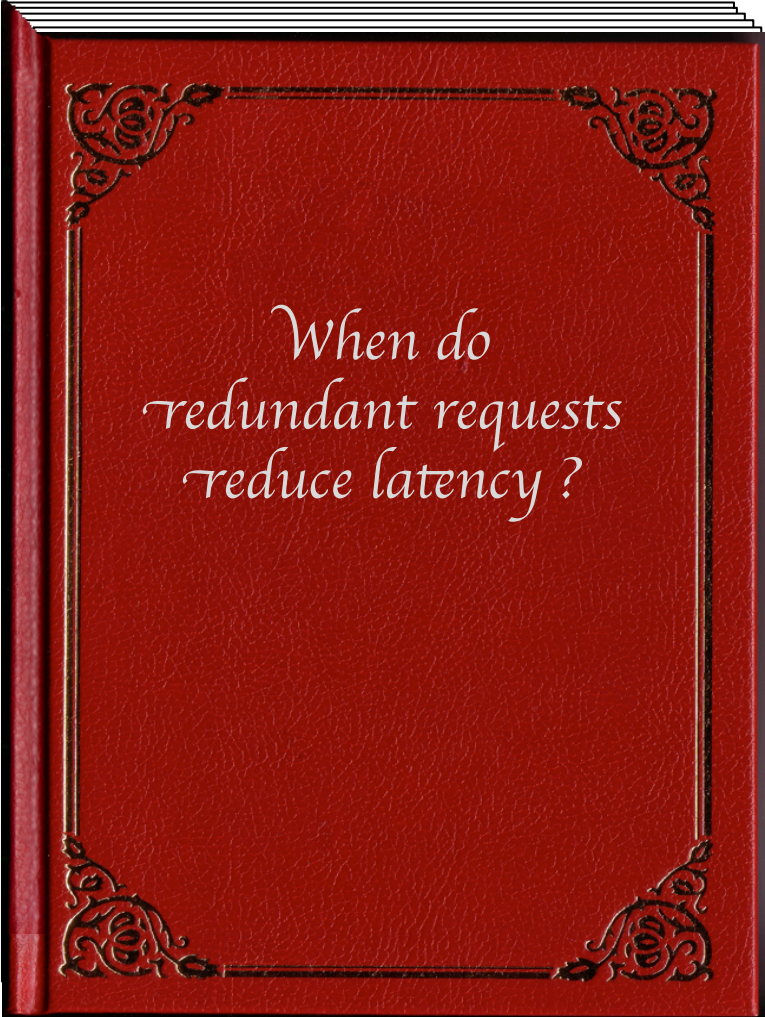
Dean & Barroso '13, Huang et al. '12, Ananthanarayanan et al. '12, Vulimiri et al. '12, Liang & Kozat, '13

SYSTEMS

- Flech et al., SIGCOMM '13
- Huang et al. ATC '12
- Dean & Barroso, Comm. of ACM '13
- Stewart et al., ICAC '13
- Liang & Kozat, Trans. on NW '13
- Vulimiri et al., HotNets '12
- Ananthanarayanan et al., HotCloud '12
- Pitkanen & Ott, MobiArch '07
- Andersen et al., NSDI '05
- Snoeren et al., SIGOPS OSR '01

⋮

Dean & Barroso '13, Huang et al. '12, Vulimiri et al. '12, Ananthanarayanan et al. '12, Liang & Kozat, '13, Flech et al. '13, Stewart et al., '13, Pitkanen & Ott '07, Andersen et al. '05, Snoeren et al. '01, ...



*When do
redundant requests
reduce latency ?*



When do redundant requests reduce latency ?

SYSTEMS

- Flech et al., SIGCOMM '13
- Huang et al. ATC '12
- Dean & Barroso, Comm. of ACM '13
- Stewart et al., ICAC '13
- Liang & Kozat, Trans. on NW '13
- Vulimiri et al., HotNets '12
- Ananthanarayanan et al., HotCloud '12
- Pitkanen & Ott, MobiArch '07
- Andersen et al., NSDI '05
- Snoeren et al., SIGOPS OSR '01

⋮

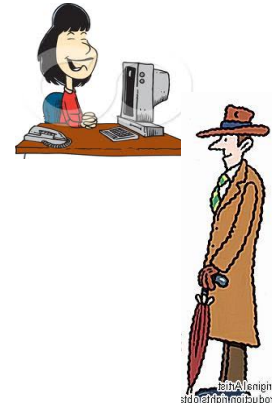
The Tail at Scale

DOI:10.1142/S087624540801784

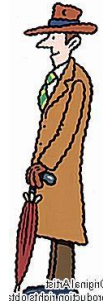
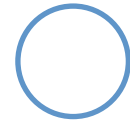
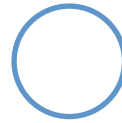
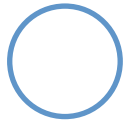
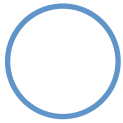
software techniques form a predictable whole out of less-predictable parts. Unit-to-unit communication, file-to-file transfer and no genuine analogies to large-scale distributed systems. Eliminating all sources of latency variability in large-scale systems is impractical, especially in shared environments. Even more recent performance fluctuations affect a significant fraction of all requests in large-scale distributed systems.

74 COMMUNICATIONS OF THE ACM | FEBRUARY 2013 | VOL. 56 | NO. 2

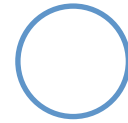
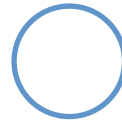
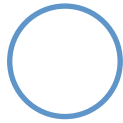
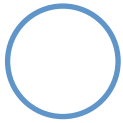
Model



parameters n , k , r



n servers (here, $n = 4$)



requests

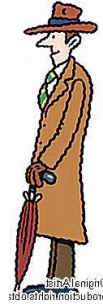
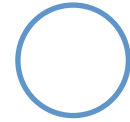
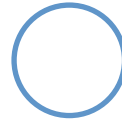
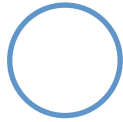
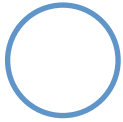
A

B

C

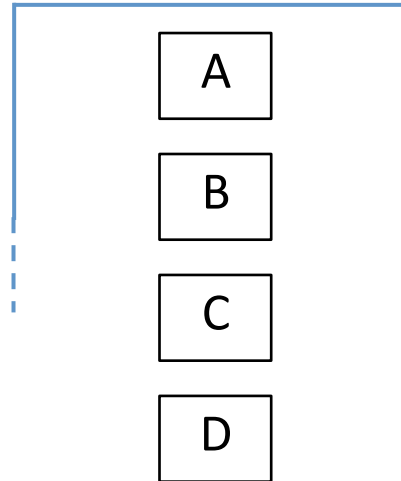
D

n servers

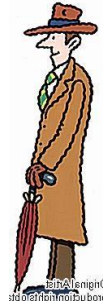
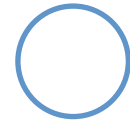
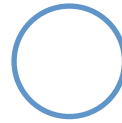
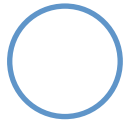
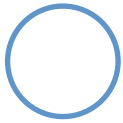


a request can be served
by *any* k *distinct* servers

here, $k = 1$

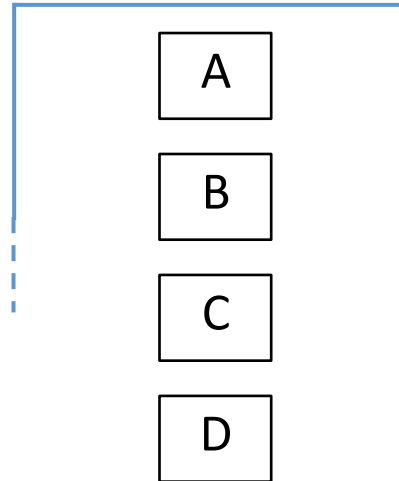


n servers



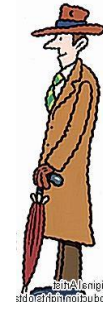
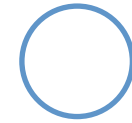
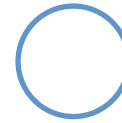
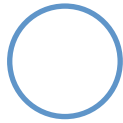
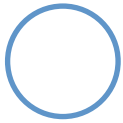
request redundantly
sent to r servers

here, $r = 2$



n servers

request served by *any* k distinct servers

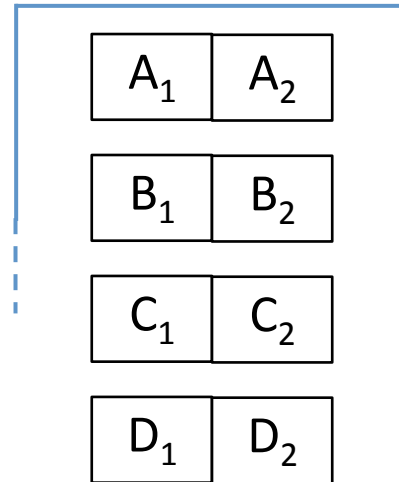


request redundantly
sent to r servers

here, $r = 2$

each request represented
as $r = 2$ jobs

any k jobs must get processed
(by distinct servers)



n servers

request served by *any* k distinct servers

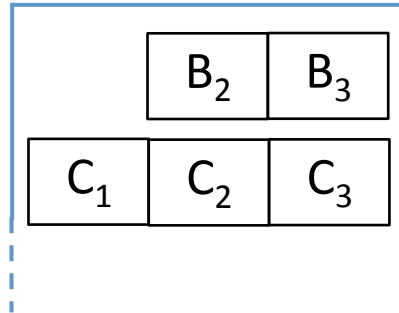
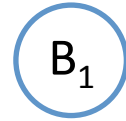
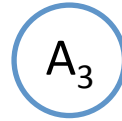
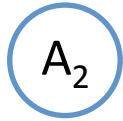
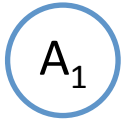
We consider FCFS Scheduling (First-come, First-served)

Example...

n servers

request served by *any* k *distinct* servers

redundantly sent to r servers

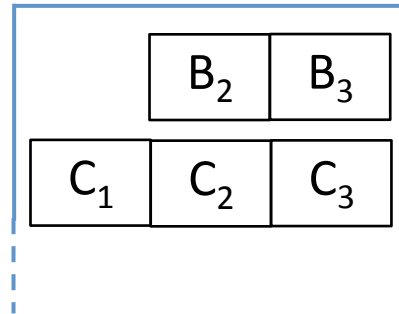
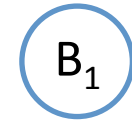
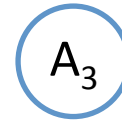
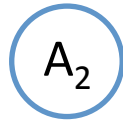
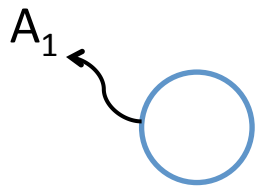


n servers

request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$



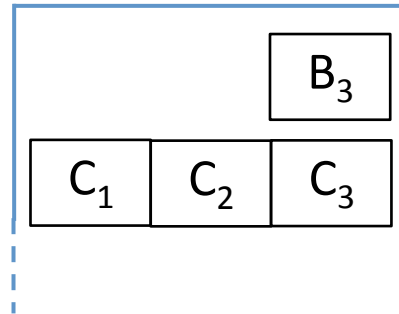
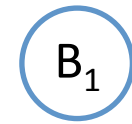
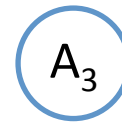
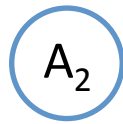
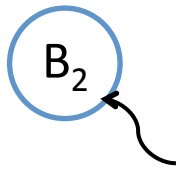
n servers

request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$

server 1 completes service



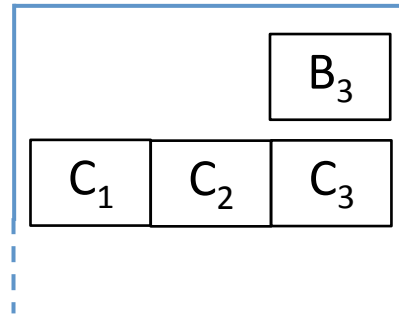
n servers

request served by *any* k *distinct* servers

redundantly sent to r servers

$n = 4, k = 2, r = 3$

B_2 goes to server 1 (FCFS)



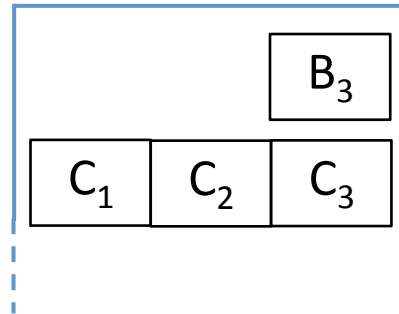
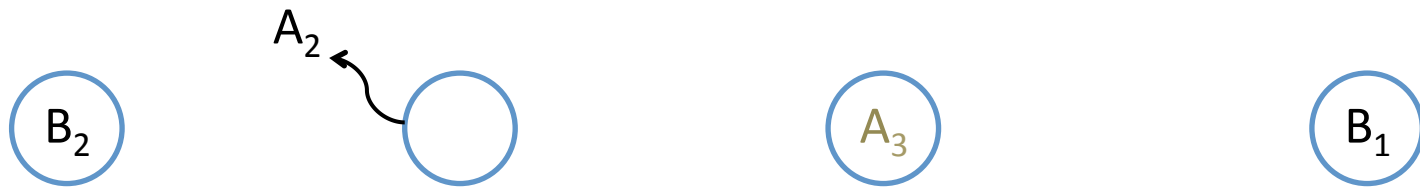
n servers

request served by *any* k *distinct* servers

redundantly sent to r servers

$n = 4, k = 2, r = 3$

server 2 completes service



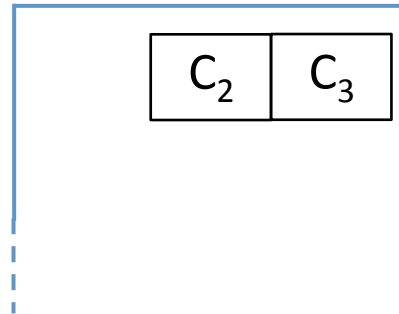
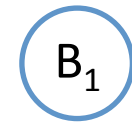
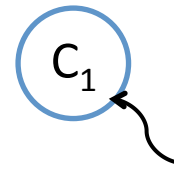
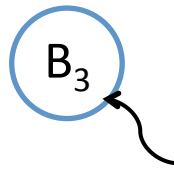
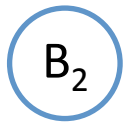
n servers

request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$

2 jobs of A served $\Rightarrow A_3$ REMOVED from system



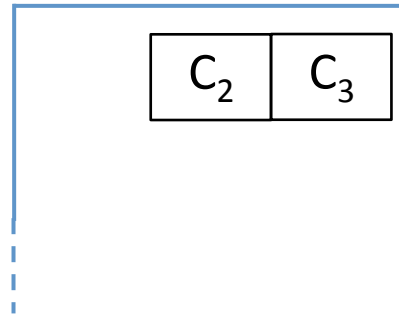
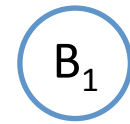
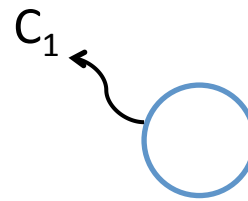
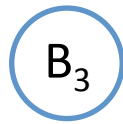
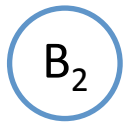
n servers

request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$

B_3 and C_1 begin to be served



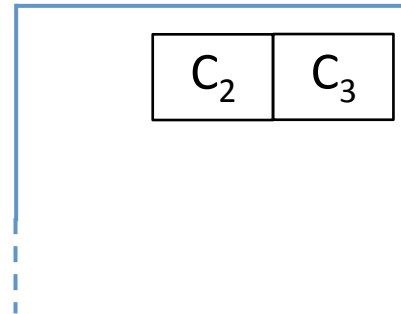
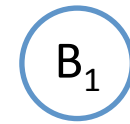
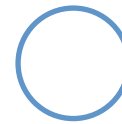
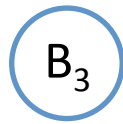
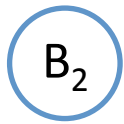
n servers

request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$

C_1 served



n servers

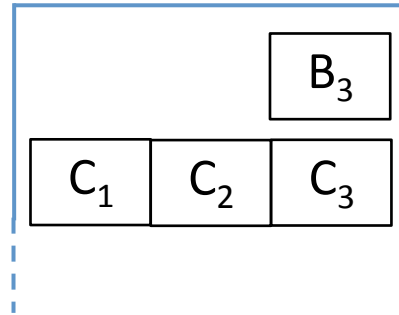
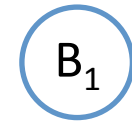
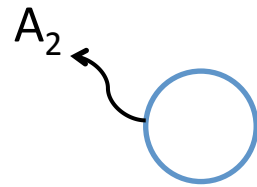
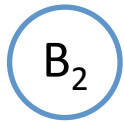
request served by *any* **k** *distinct* servers

redundantly sent to **r** servers

$n = 4, k = 2, r = 3$

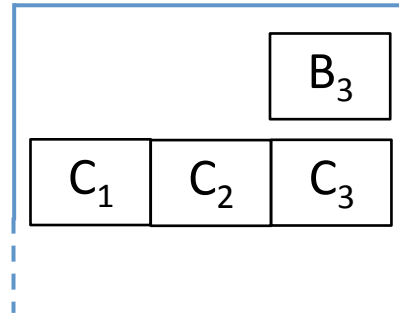
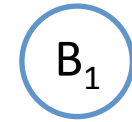
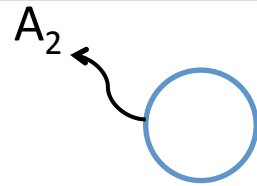
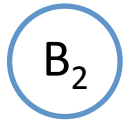
server 3 has already served C_1
 \Rightarrow cannot serve C_2 or C_3

Results



arbitrary arrival process

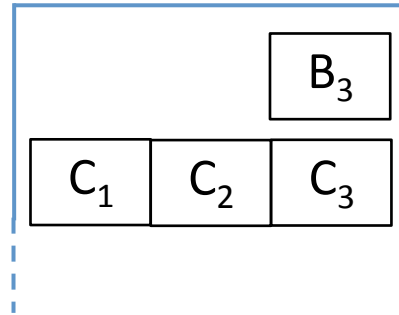
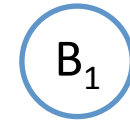
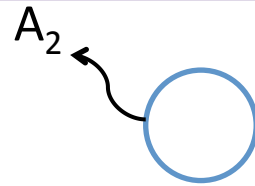
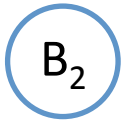
i.i.d. memoryless service



arbitrary arrival process

i.i.d. memoryless service

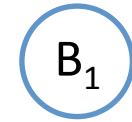
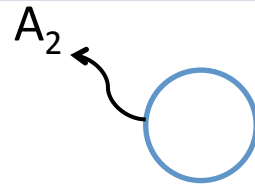
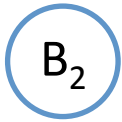
no cost of removal



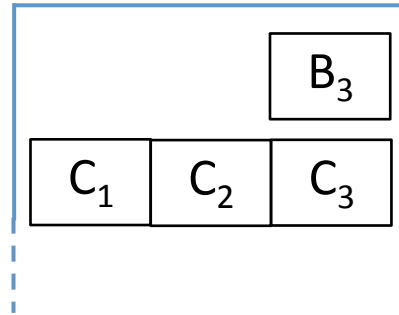
arbitrary arrival process

i.i.d. memoryless service

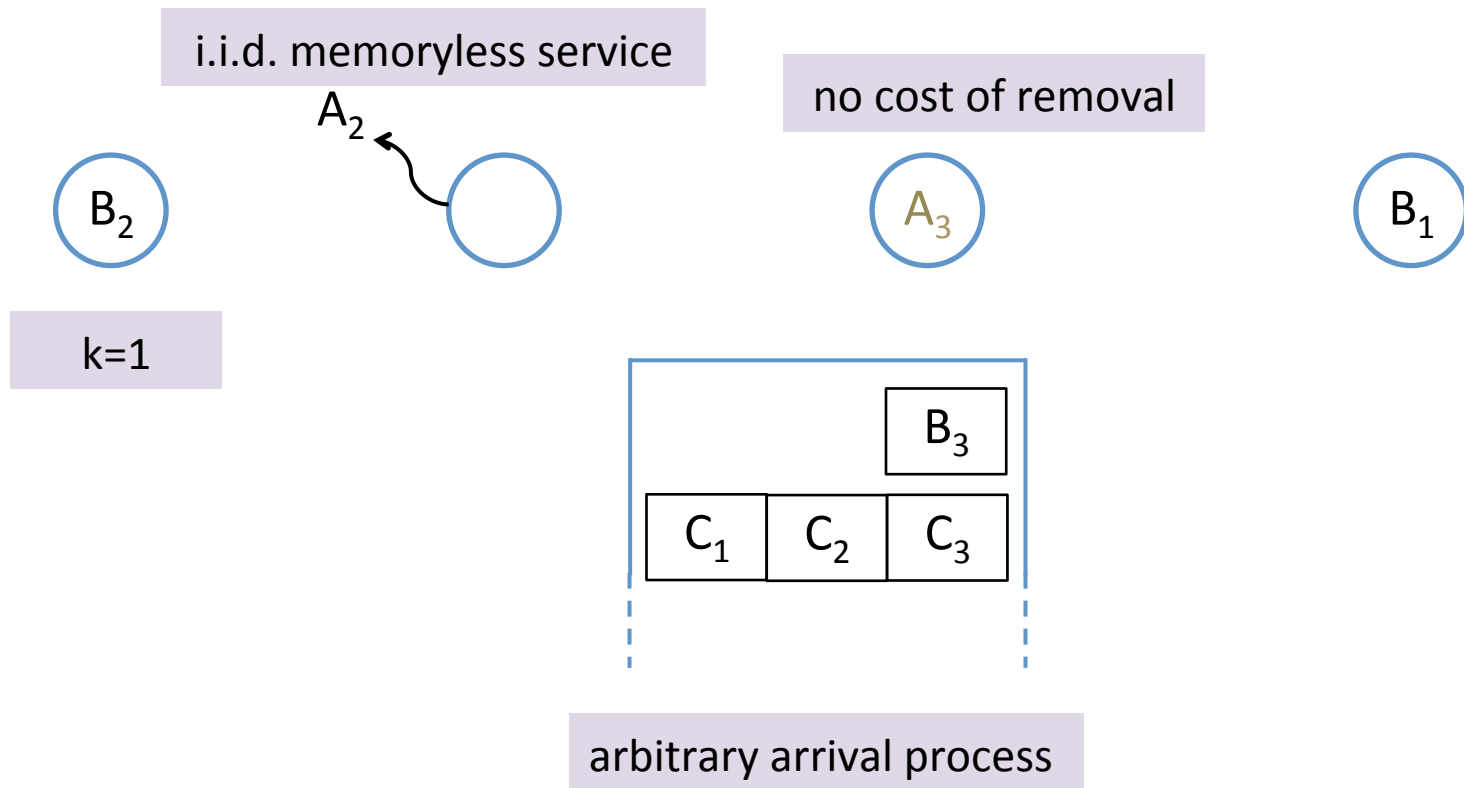
no cost of removal



$k=1$

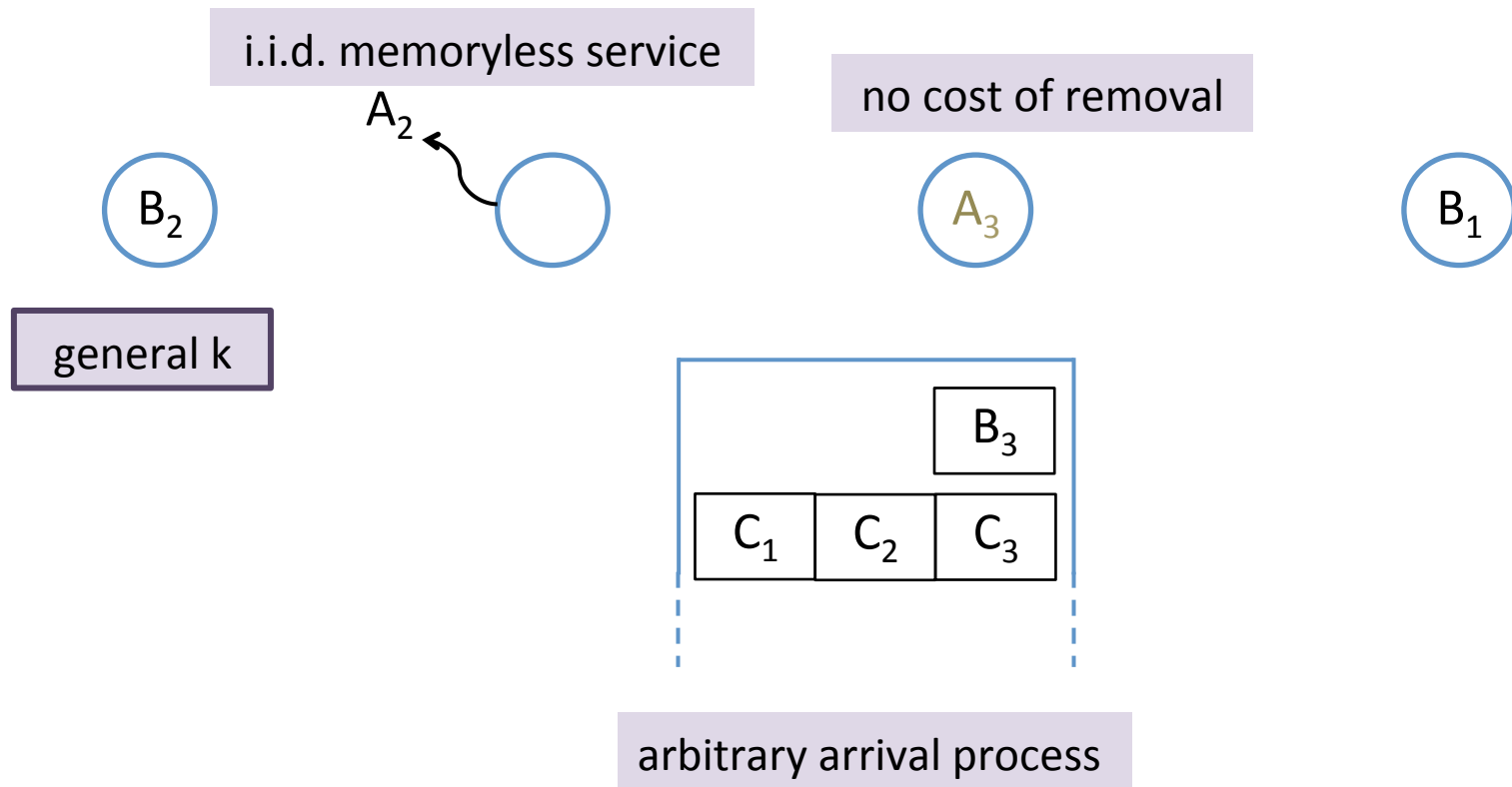


arbitrary arrival process



Theorem 1

Higher $r \Rightarrow$ lower average latency



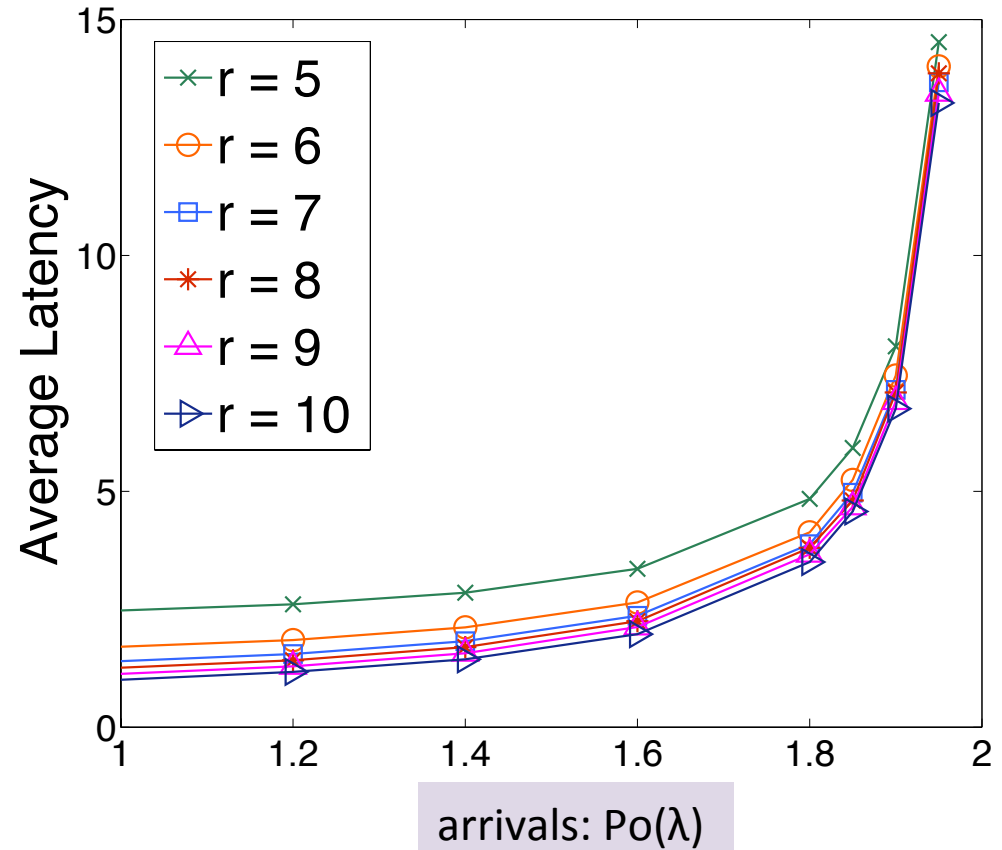
Theorem 2

$r = n$ minimizes average latency among all possible redundant-request policies.

i.i.d. memoryless service: $\exp(1)$

no cost of removal

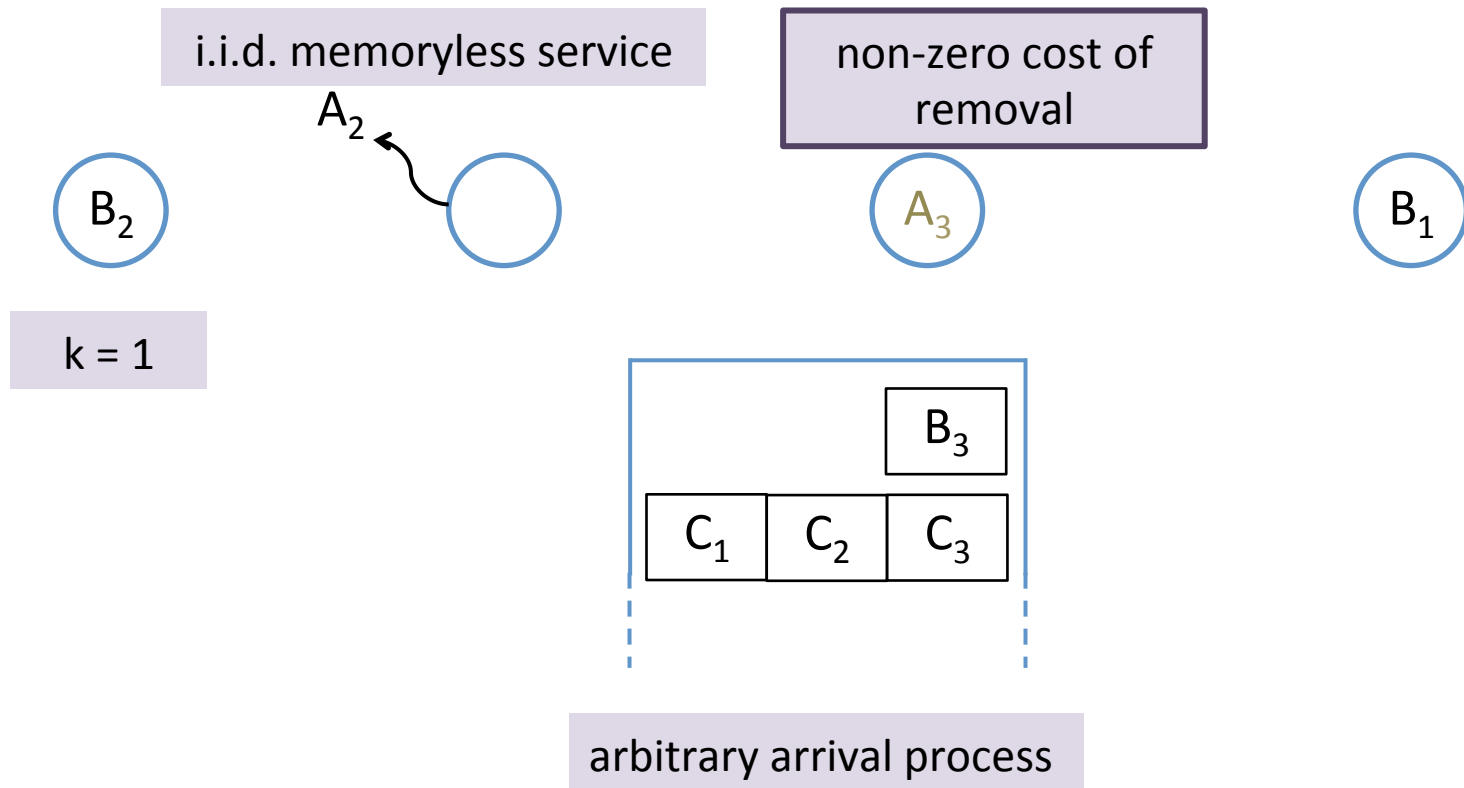
$n = 10$
 $k = 5$



Simulations

Theorem 2

$r = n$ minimizes average latency among all possible redundant-request policies.



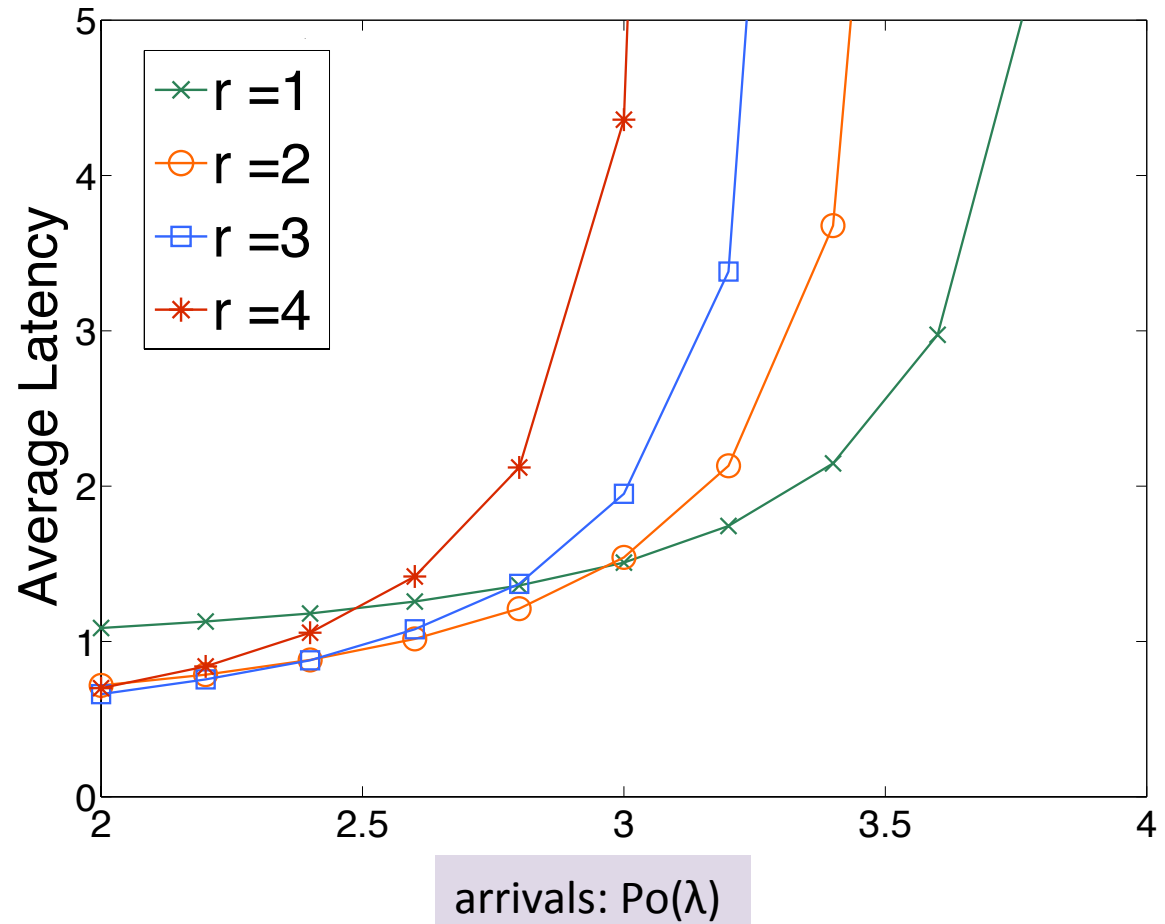
Theorem 3

$r = k$ minimizes average latency under high loads.

i.i.d. memoryless service: $\exp(1)$

removal entails wait of $\exp(10)$

$n = 4$
 $k = 1$



Simulations

Theorem 3

$r = k$ minimizes average latency under high loads.

Heavy-everywhere distribution

Memoryless:

$$P(X > s+t | X > t) = P(X > s) \quad \forall s \geq 0, t > 0$$

Heavy-everywhere distribution

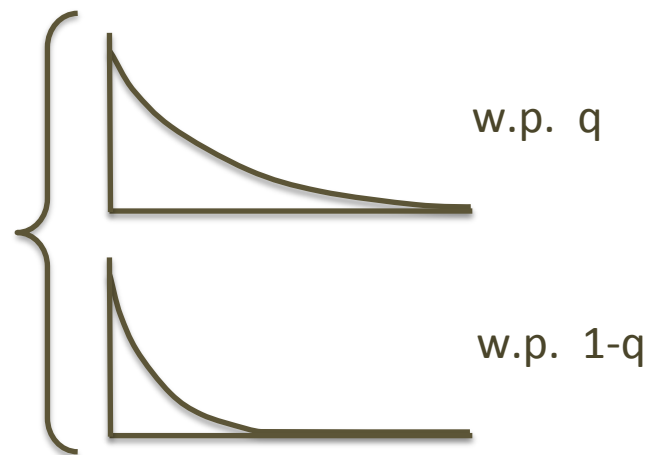
$$P(X > s+t | X > t) \geq P(X > s) \quad \forall s \geq 0, t > 0$$

Heavy-everywhere distribution

$$P(X > s+t | X > t) \geq P(X > s) \quad \forall s \geq 0, t > 0$$

Example:

mixture of independent exponentials



Light-everywhere distribution

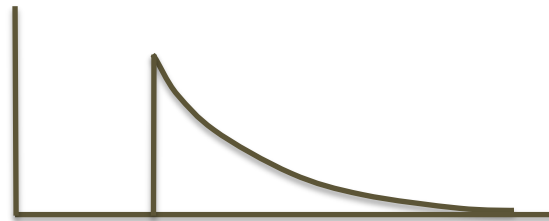
$$P(X > s+t | X > t) \leq P(X > s) \quad \forall s \geq 0, t > 0$$

Light-everywhere distribution

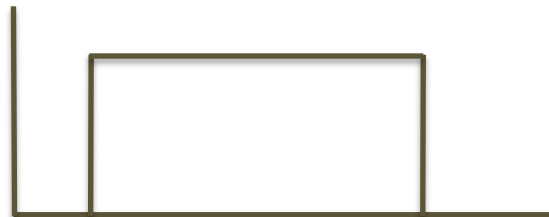
$$P(X > s+t | X > t) \leq P(X > s) \quad \forall s \geq 0, t > 0$$

Examples:

constant + exponential

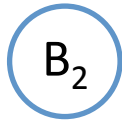


uniform

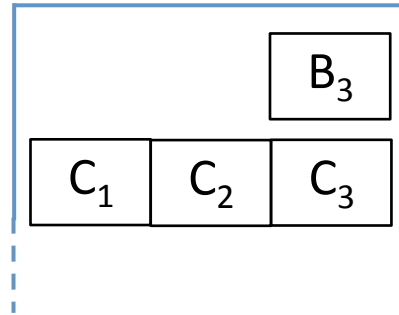
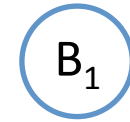
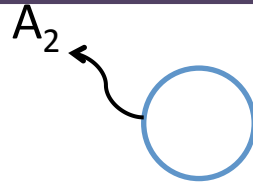


i.i.d. heavy-everywhere service

no cost of removal



$k = 1$



arbitrary arrival process

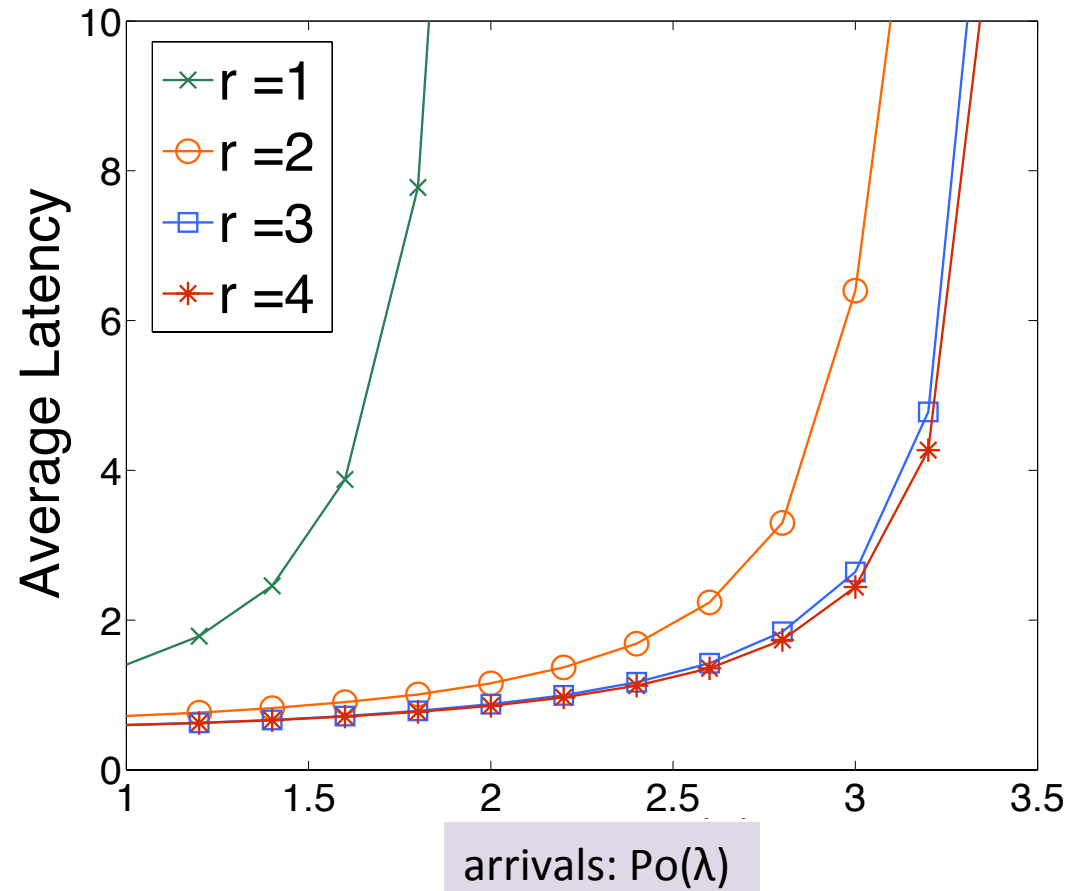
Theorem 4

$r = n$ minimizes average latency under high loads.

service: mixture of $\exp(1)$ & $\exp(10)$

no cost of removal

$n = 4$
 $k = 1$



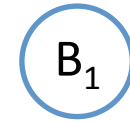
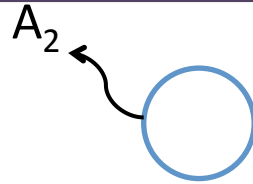
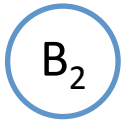
Simulations

Theorem 4

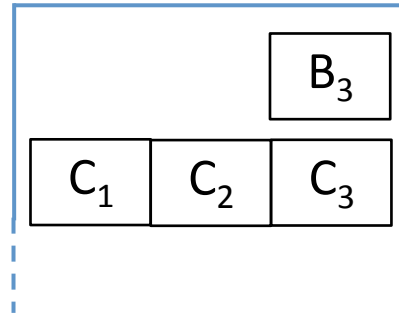
$r = n$ minimizes average latency under high loads.

i.i.d. light-everywhere service

any cost of removal



$k = 1$



arbitrary arrival process

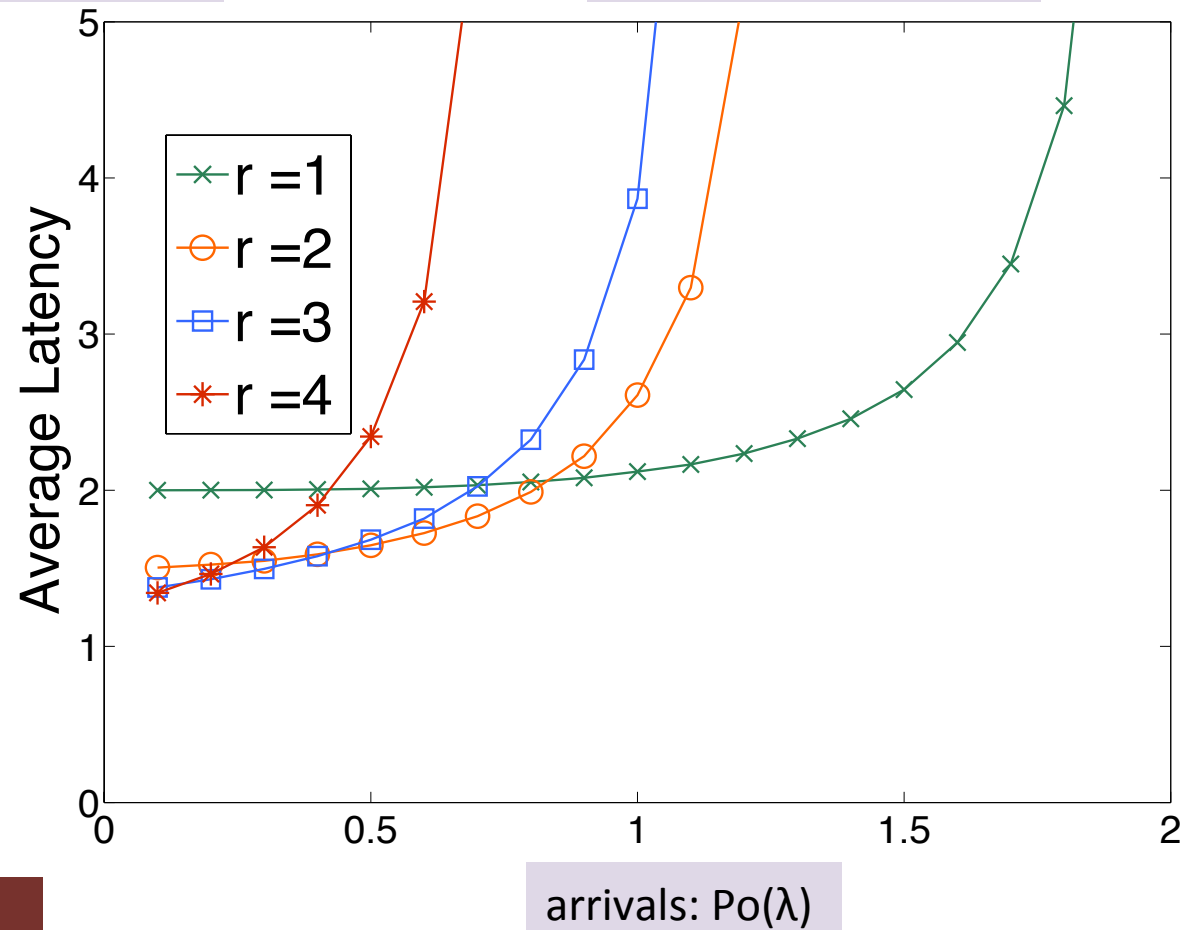
Theorem 5

$r = k$ minimizes average latency under high loads.

$$\text{service} = 1 + \exp(1)$$

no cost of removal

$n = 4$
 $k = 1$

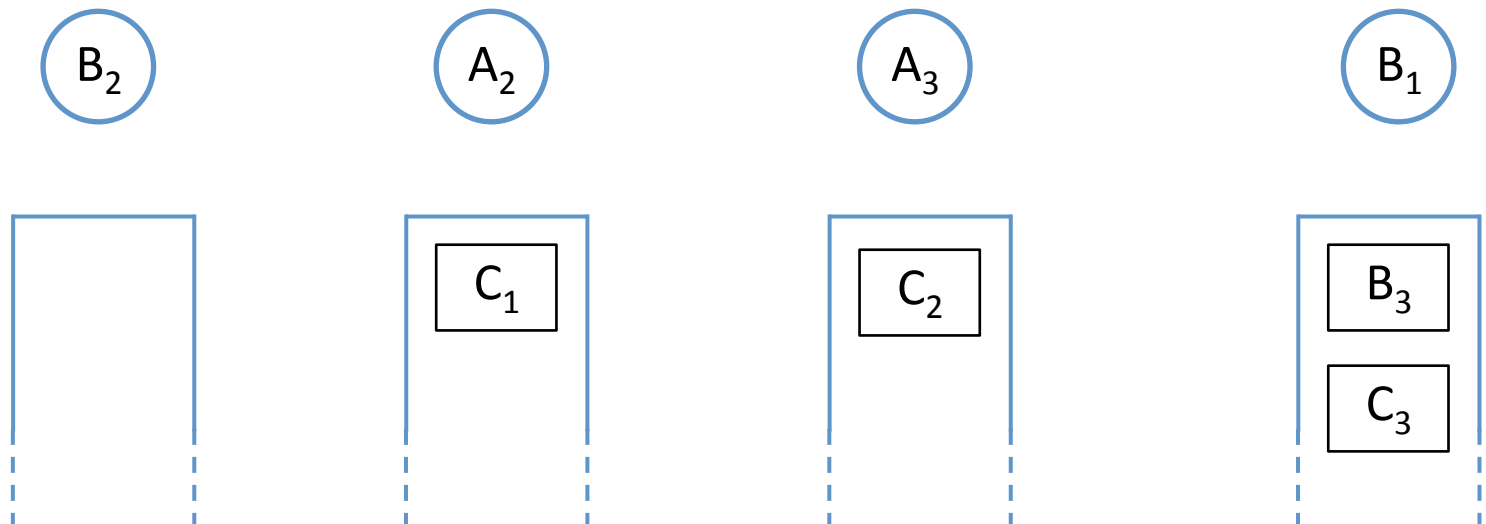


Simulations

Theorem 5

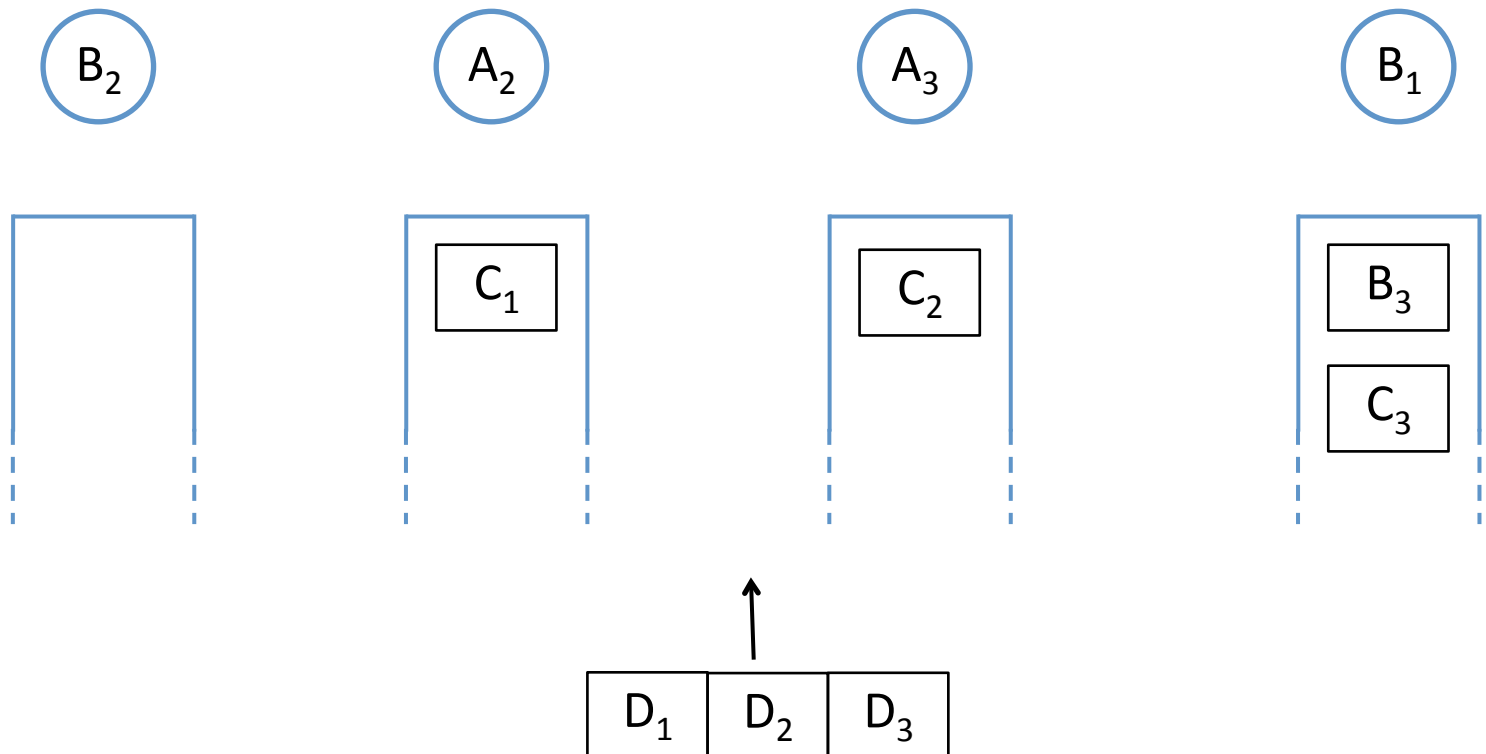
$r = k$ minimizes average latency under high loads.

Distributed Buffers



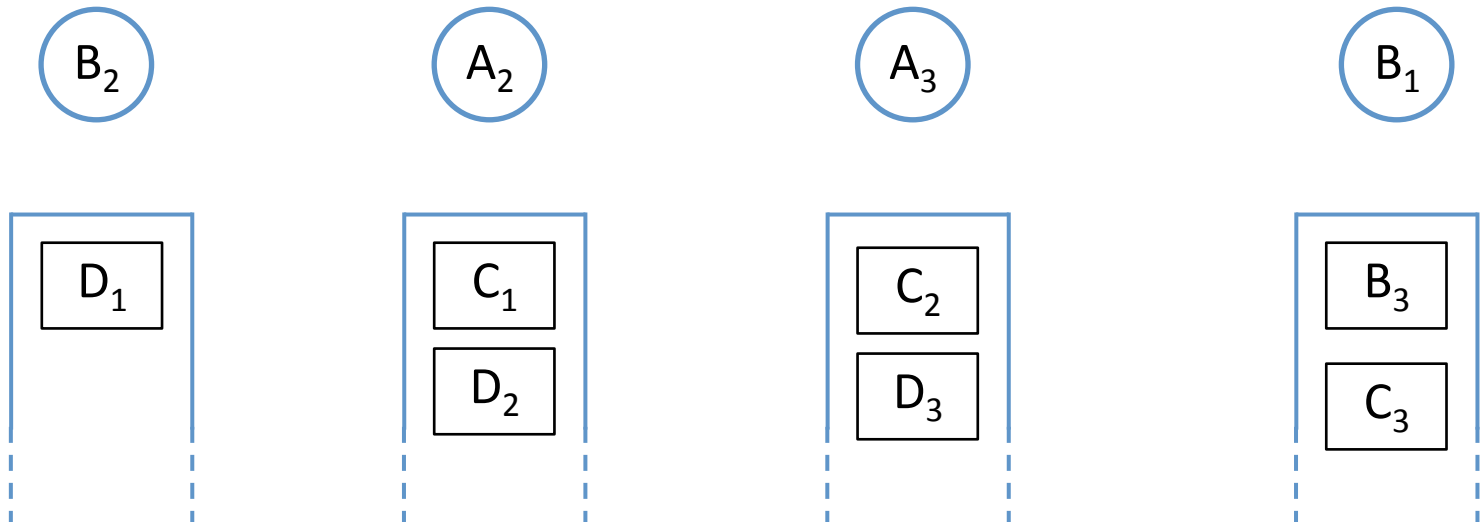
Distributed Buffers

Request must be assigned to r buffers **upon arrival**

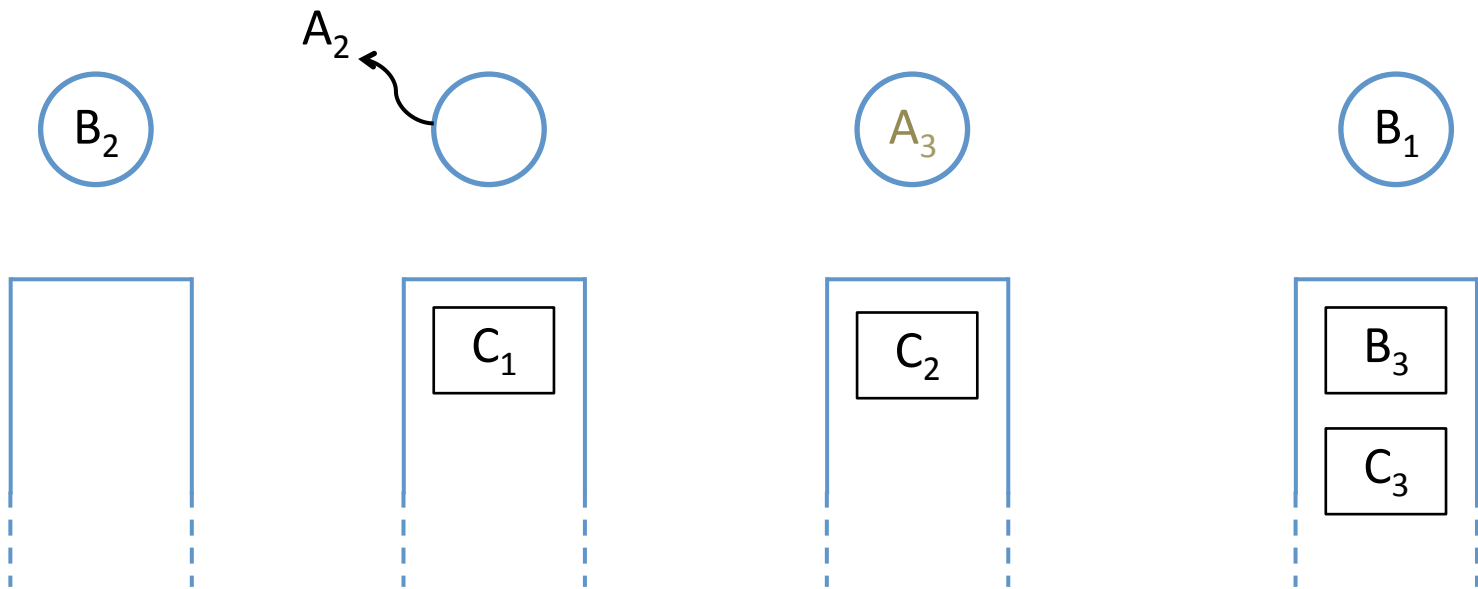


Distributed Buffers

Request must be assigned to r buffers **upon arrival**



distributed buffers



Theorem 6

i.i.d. memoryless service: $r = n$ minimizes average latency.

Theorem 7

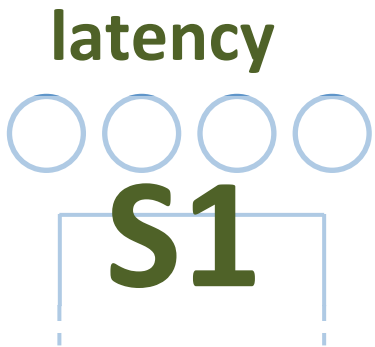
i.i.d. heavy-everywhere service: $r = n$ minimizes average latency under high loads.

Theorem 8

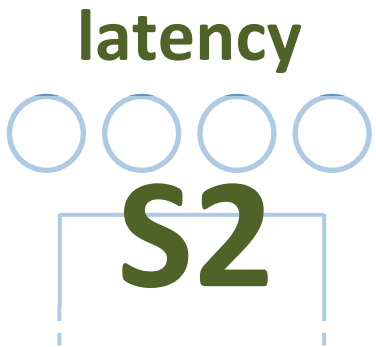
i.i.d. light-everywhere service: $r = k$ minimizes average latency under high loads.

General Proof Technique

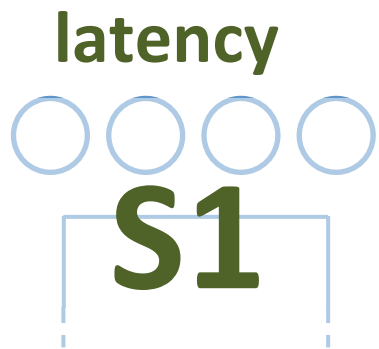
Wish to show:



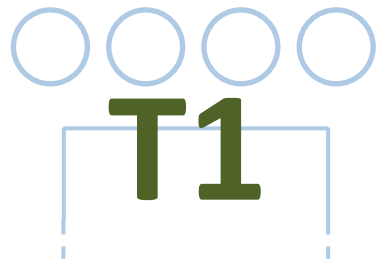
?



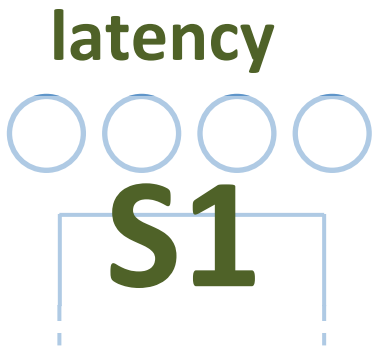
Wish to show:



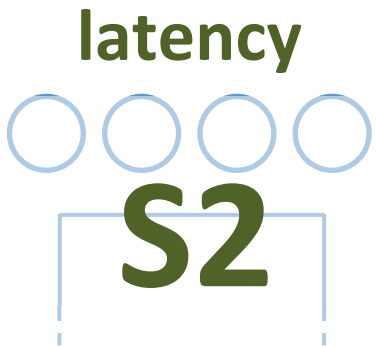
Construct
hypothetical
systems



Wish to show:

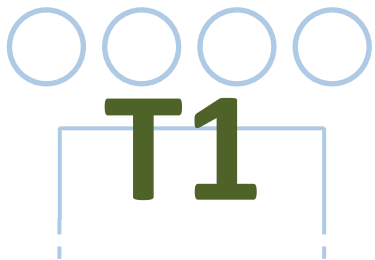


?

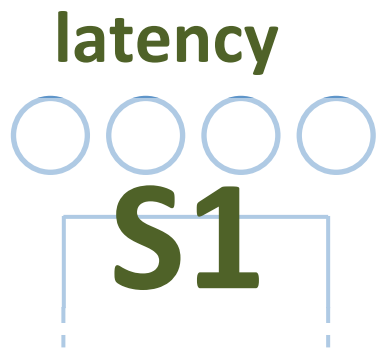


T1 is statistically
identical to or
better than S1

Construct
hypothetical
systems



Wish to show:

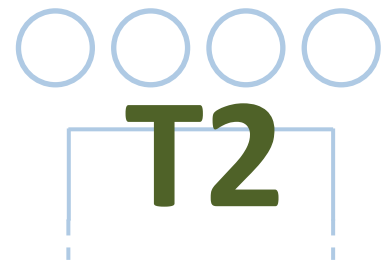
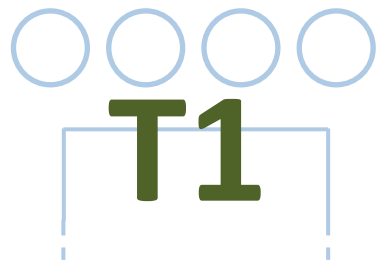


?

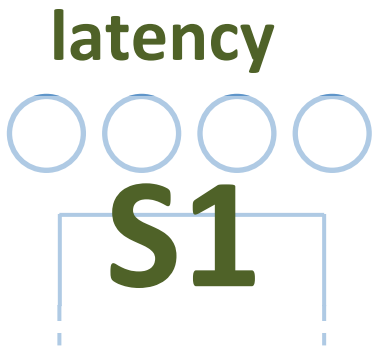


IV

Construct
hypothetical
systems

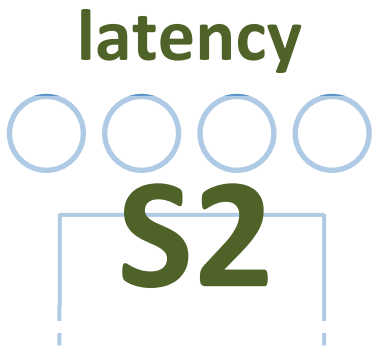


Wish to show:



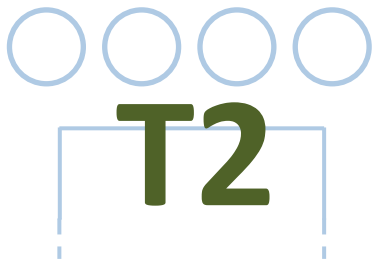
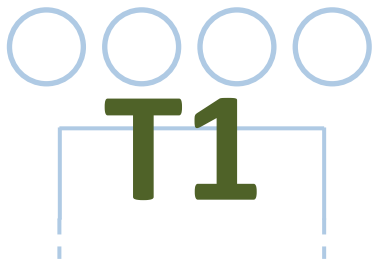
?

>

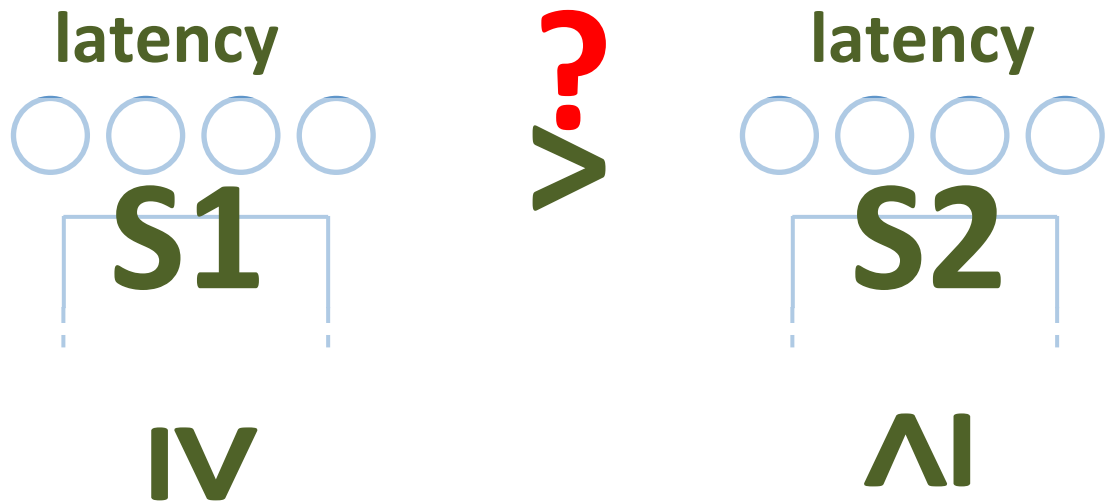


T2 is statistically
identical to or
worse than S2

Construct
hypothetical
systems



Wish to show:



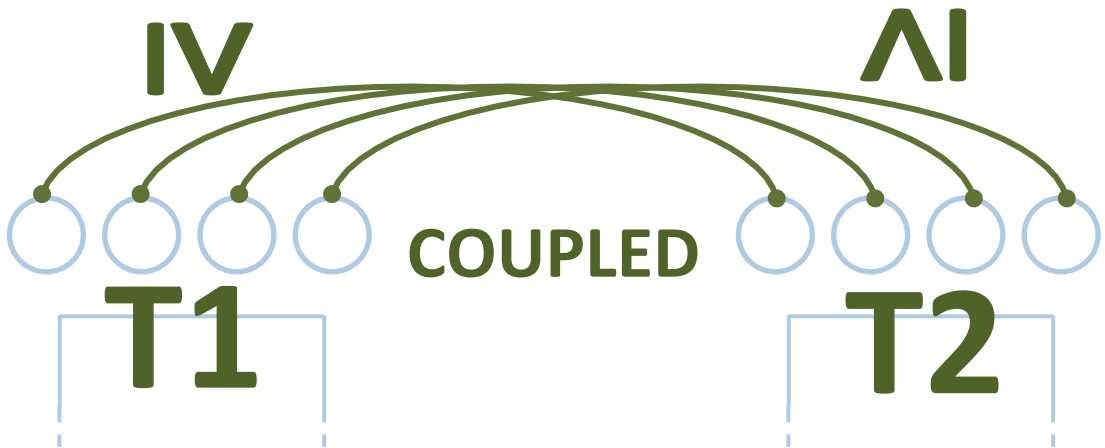
Construct
hypothetical
systems



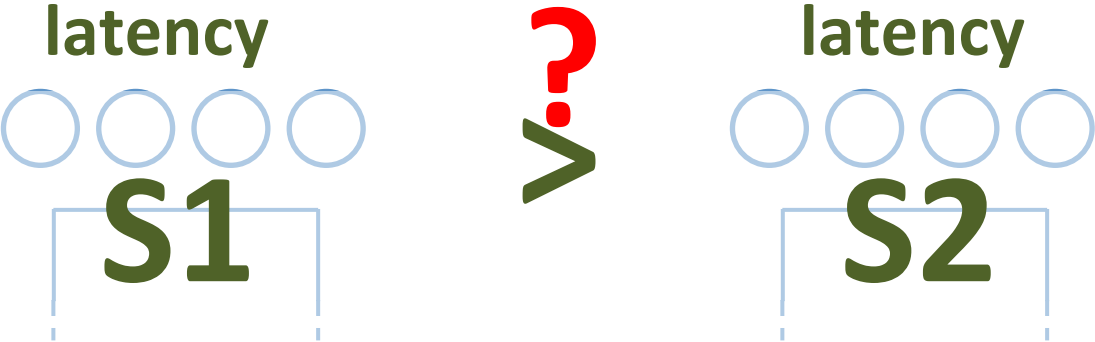
Wish to show:



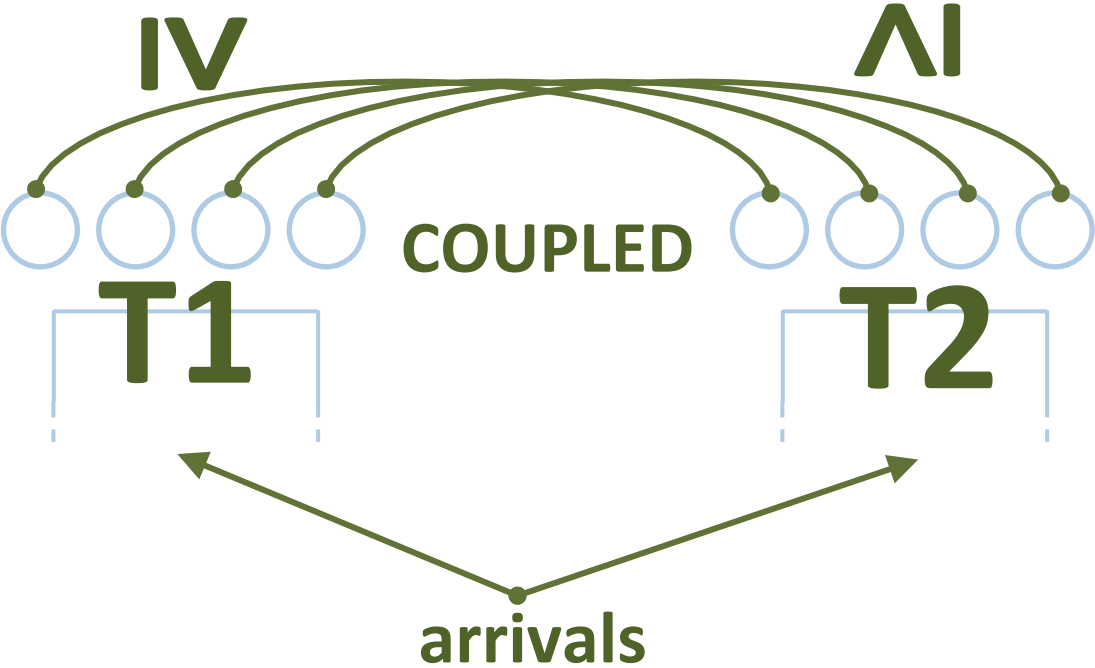
Construct
hypothetical
systems



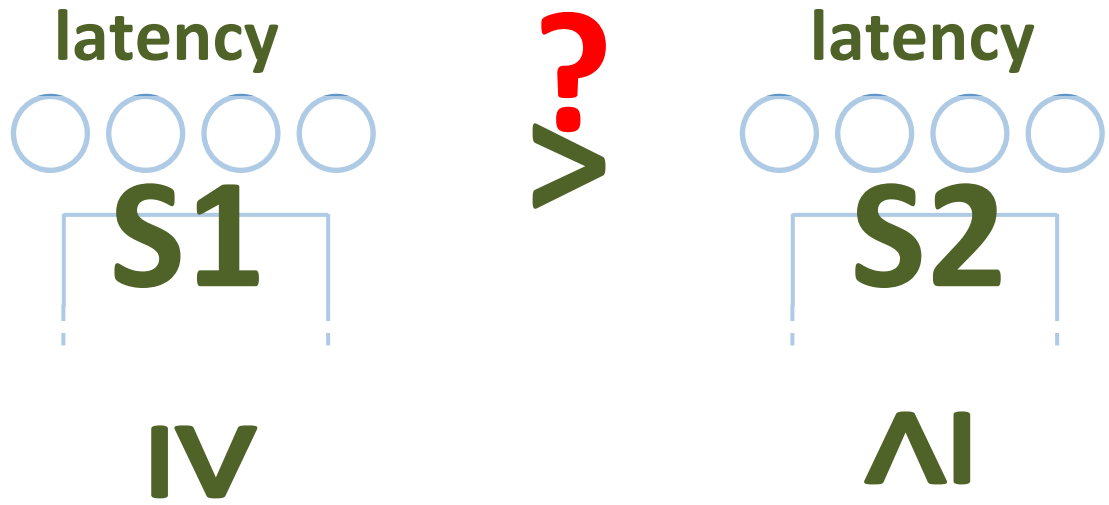
Wish to show:



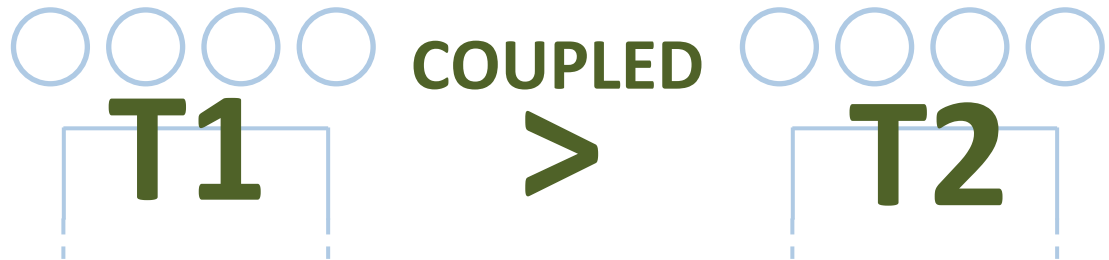
Construct
hypothetical
systems



Wish to show:



Construct
hypothetical
systems



Show that at every point in time,
T2 is in a better state than T1

Summary

- Redundant requests empirically observed to help in several settings, hurt in some others
- We aim for a theoretical characterization
- Propose a basic model and show:

n	k	arrival	service	buffers	removal cost	load	result
any	1	any	iid memoryless	centralized	0	any	higher r better
any	any	any	iid memoryless	centralized	0	any	$r = n$ optimal
any	1	any	iid heavy-everywhere	centralized	0	high	$r = n$ optimal
any	1	any	iid light-everywhere	centralized	any	high	$r = 1$ optimal
any	1	any	iid memoryless	centralized	>0	high	$r = 1$ optimal
any	any	any	iid memoryless	distributed	0	any	$r = n$ optimal
any	1	any	iid heavy-everywhere	distributed	0	high	$r = n$ optimal
any	1	any	iid light-everywhere	distributed	any	high	$r = 1$ optimal

- Proof techniques of independent interest

Open problems

- Simulations show redundant-requests stop helping beyond certain **threshold: analytical characterization ?**
- Requests or the servers are **heterogeneous or correlated ?**
- If allowed to choose “r” adaptively, **optimal redundant-requesting policy ?**
- Settings when a request can be processed by only **specific servers**
- **Other metrics:** tails of latency; quantification of amount of gains ?
(Joshi et al.: bounds when sending to all n)