LUND UNIVERSITY

Modeling and Control for Improved Predictability of Cloud Applications

Berner, Tommi

2022

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Berner, T. (2022). *Modeling and Control for Improved Predictability of Cloud Applications*. [Doctoral Thesis (compilation), Department of Automatic Control]. Department of Automatic Control, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Modeling and Control for Improved Predictability of Cloud Applications

Tommi Berner

LUND
UNIVERSITY

Department of Automatic Control

PhD Thesis TFRT-1136
ISBN 978-91-8039-233-4 (print)
ISBN 978-91-8039-234-1 (web)
ISSN 0280–5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2022 by Tommi Berner. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2022

Abstract

Cloud computing has emerged as a key technology in the latest decade and continues to be applied to manage the computing needs of new domains. As a result, the requirements on predictable behavior in the cloud increase, thus the hosted applications need to be recognized as both fault-tolerant and responsive even under difficult conditions.

In this thesis, new modeling methods and decision-making strategies are presented with the goal of increasing the predictability of cloud applications. The methods can be divided into two tracks, using concepts from control theory and queuing theory respectively. The control-theoretical method track utilizes the concept of *graceful degradation* as an enabling actuator. In the context of server control, a novel dynamic model for queue lengths is proposed, as well as a cascaded structure for response time control. Additionally, interactions between decision-making strategies at different layers in the cloud infrastructure are discussed, including an interpretation of the popular Join-Shortest-Queue (JSQ) load-balancing strategy as a queue length controller.

The queuing-theoretical track utilizes the concept of request *cloning* to increase the predictability of applications replicated across multiple servers. A criterion for *synchronized service* is formalized, which enables a dramatic simplification of modeling of applications subject to cloning, without requiring any further assumptions on neither queuing disciplines nor on the statistical distributions involved. Furthermore, model error bounds are derived for server systems that break the synchronized service criterion. It is shown that imperfections that can arise during implementation, only slightly affect the accuracy of the model. Finally, an intuitive explanation is given for why the popular JSQ load-balancing strategy acts as a *service synchronizer*, that allows for accurate, approximate modeling of the complicated scenario of unrestricted request cloning across replicated servers where the JSQ strategy is used for load-balancing.

While there are differences in the modeling approaches between the two separate method tracks, they both share common properties that run

throughout the thesis. First, the majority of the involved techniques revolve around finding design choices that enable simplification, without limiting the applicability of the solutions. Second, many of the strategies presented in the thesis apply concepts and structures traditionally used in different domains, which often requires the problems to be viewed from a slightly different angle. The proposed models and methods from both tracks are evaluated in a simulated cloud environment, composed of a discrete-event simulator implemented in a request-by-request fashion, independent of the proposed methods in this thesis.

Acknowledgements

I would like to begin by thanking my supervisors, Karl-Erik, Martina and Maria. Karl-Erik, my main supervisor, for interesting talks about both MFF and our research, and for providing guidance and support without applying unnecessary pressure. Martina, for giving me both high-level advice as well as valuable input regarding technical questions, and Maria for helping me getting started with my PhD during the first years.

I also want to thank everyone at the Department of Automatic Control for contributing to a welcoming and inspiring work environment. During my years at the department, I always felt that there was someone to ask if I was stuck, whatever the problem was. I would specially like to thank the colleagues that I worked close together with, both within WASP and the cloud research group, including Jonas, Manfred, Victor, Gautham, Alexandre, Per, Marcus and Johan. A special thanks also to Jacob for all chess and sports related discussions we had in our office.

Thanks also to the administrative staff, Eva, Ingrid, Mika, Cecilia and Monika, for help and support with all kinds of work-related things. The same goes for the technical staff, Anders N, Anders B, Pontus and Leif, thank you for helping me with all the hardware and software related problems I had during my time at the department.

I would like to thank my parents and my brother for all support during the years that led me here. Last but not least, I would also like to thank my wife Josefin for her invaluable support in all kinds of matters, and our son Anton for his ability to make me stop thinking about work. It would not be the same without you.

Financial support

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Contents

Acronyms **9**
1. Introduction **10**
1.1 Thesis Outline 11
1.2 Contributions of the Thesis 12
2. Background **16**
2.1 Cloud Computing 16
2.2 Queuing Theory 24
2.3 Control Concepts 30
3. Improving Cloud Application Predictability **37**
3.1 Simulation Environment 37
3.2 Control-Theoretical Methods 38
3.3 Queuing-Theoretical Methods 46
3.4 Common Themes 50
4. Future Work **52**
Bibliography **53**
Paper I. Brownout^{CC}: Cascaded Control for Bounding the Response Times of Cloud Applications **59**
1 Introduction 60
2 The brownout approach 61
3 The Brownout^{CC}approach 63
4 Evaluation 71
5 Conclusion and future work 77
References 78
Paper II. Improved Dynamic Modeling for Controlled Server Queues **83**
1 Introduction 84
2 Background and Related Work 86
3 Model 89
4 Model Evaluation 96

5 Control Design . . . 102
6 Control Evaluation . . . 105
7 Conclusion . . . 109
References . . . 111

Paper III. Cloud Application Predictability through Integrated Load-Balancing and Service Time Control 115
1 Introduction . . . 116
2 Problem Statement . . . 118
3 Proposed Solution . . . 121
4 Experimental Validation . . . 131
5 Related work . . . 136
6 Conclusion . . . 136
References . . . 137

Paper IV. Modeling of Request Cloning in Cloud Server Systems using Processor Sharing 143
1 Introduction . . . 144
2 Synchronized Model . . . 146
3 Examples . . . 150
4 Applications . . . 151
5 Non-Synchronized Service . . . 157
6 Evaluation . . . 164
7 Related Work . . . 167
8 Conclusion . . . 170
References . . . 171

Paper V. Towards Performance Modeling of Speculative Execution for Cloud Applications 175
1 Introduction . . . 176
2 Model . . . 177
3 Evaluation . . . 180
4 Conclusion . . . 182
References . . . 182

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CDF	Cumulative Distribution Function
CFPS	Combined FIFO and PS, equivalent to LPS
CoC	Cancel-on-Complete (cloning)
CoS	Cancel-on-Start (cloning)
FaaS	Function-as-a-Service
FCFS	First-Come First-Served
FIFO	First-In First-Out, equivalent to FCFS
IaaS	Infrastructure-as-a-Service
JSQ	Join Shortest Queue
LB	Load Balancer
LPS	Limited Processor Sharing
MIMO	Multiple-Input Multiplie-Output (system)
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform-as-a-Service
PDF	Probability Density Function
PI	Proportional and Integral (controller)
PID	Proportional, Integral and Derivative (controller)
PS	Processor Sharing
PSFFA	Pointwise Stationary Fluid Flow Approximation
SaaS	Software-as-a-Service
SISO	Single-Input Single-Output (system)
SQF	Shortest Queue First, equivalent to JSQ
VM	Virtual Machine
WASP	Wallenberg AI, Autonomous Systems and Software Program

1

Introduction

Cloud computing[1] is a concept where computational resources and storage are provided as a utility, i.e., where cloud users pay for their usage from a seemingly infinite pool of resources [Barroso and Hölzle, 2009]. However, behind the scenes there is of course a limitation on the amount of physical machines that the cloud providers can maintain in their data centers, and there are many components involved that can fail [Dean and Barroso, 2013]. Furthermore, the cloud users share the physical resources through virtualization which creates additional complexity, both in terms of performance and security [Xing and Zhan, 2012]. This makes the task of providing cloud computing as a service difficult. Today, there exists many different service models that define what type of computing services the cloud providers offer. The most common examples include: Infrastructure as a Service (IaaS); Platform as a Service (PaaS); and Software as a Service (SaaS), stated in an increasing level of abstraction [Mell and Grance, 2011].

The scope of this thesis is limited to concern cloud users that host applications using the IaaS concept. Specifically, the thesis is focused on the computational performance of the application, i.e., disregarding, e.g., storage, network and security. In this setting, the computational part of the application is deployed on the cloud using either virtual machines [Smith and Nair, 2005] or containers [Soltesz et al., 2007]. At the abstraction level used in this thesis, both these virtualization techniques are considered as identical and the computation units in the cloud are denoted as *servers*. The types of cloud applications considered in this thesis are restricted to end-user facing applications that receive requests and serve responses. They are deployed in a single server, but can of course be replicated if necessary. An example could be a simple webserver that hosts a small store. However, this application definition does not comply very well with the increasingly popular microservices concept, where a single cloud application can be split into tens or hundreds of services, all deployed on separate servers [Newman, 2021]. On the other

[1] For more information regarding the cloud concepts, the reader is referred to Chapter 2.

hand, the application definition in this thesis could then be used to represent one of the involved services, instead of a complete monolithic application.

As the task of providing well-behaved servers is difficult for the cloud provider, cloud applications need to be designed and managed with respect to these inherent uncertainties, i.e., they need to be fault-tolerant. Additionally, the end-user behavior can vary, both predicatively over time as well as due to sudden events. To cope with these changes it is possible to overprovision the cloud deployment, i.e., to deploy for peak loads. This is, however, an expensive and wasteful solution [Armbrust et al., 2010]. A better option is to change the service rate of the application, e.g., by dynamically scaling the deployment or by graceful degradation, where the quality of the response is temporarily sacrificed for speed [Jalaparti et al., 2013].

In this thesis, cloud application *predictability* is defined as the ability for an application to be both fault-tolerant and responsive, i.e., providing short response times with low variance even under changing conditions. An important measure that needs to be kept low to maintain predictability is the tail latency, which represents the response times of the slowest requests, i.e., the tail of the distribution.

The aim of this thesis is to use modeling concepts and decision-making from both queuing theory and control theory, in order to improve the predictability of cloud applications. The main concepts involved include control methods using graceful degradation techniques, but also queuing theoretic modeling of request cloning, where multiple copies of the same request are issued simultaneously to the servers. Together, the methods included in this thesis all strive towards the same goal; to ensure fault-tolerant and responsive applications in an uncertain cloud environment.

1.1 Thesis Outline

This thesis is written as a collection of papers, with the following outline of the kappa:

Chapter 1 - Introduction

The introductory chapter describes the motivation and aim of the thesis, and introduces the main contributions together with the included papers.

Chapter 2 - Background

The second chapter presents important concepts within cloud computing, queuing theory and control theory that are used and referenced throughout the thesis. This background chapter is completely based on previous knowledge.

Chapter 3 - Improving Cloud Application Predictability

This chapter presents the main goal and contributions of the thesis. The assumed cloud setting is described, and the simulation environment used throughout the thesis is introduced. Additionally, the proposed modeling and decision-making strategies are presented, divided into two method tracks: (i) control-theoretical; and (ii) queuing-theoretical. Finally, common themes from both tracks are identified and discussed.

Chapter 4 - Future work

The final chapter gives suggestions for where future improvements can be made, regarding both the strategies proposed as well as the evaluation environment.

1.2 Contributions of the Thesis

The main contributions are novel models and decision-making strategies for increased predictability of cloud applications, using ideas and concepts from two method tracks: (i) control-theoretical; and (ii) queuing-theoretical. The contributions for each paper are shortly described in the remainder of this section. The control-theoretical method track is composed by Papers I-III and the queuing-theoretical by Papers IV-V. Notice that Tommi Berner has changed his surname from Nylander to Berner during the course of his PhD-studies. Both surnames are used in the papers below.

Paper I

Nylander, T., C. Klein, K.-E. Årzén, and M. Maggio (2018). "Brownout$^{\mathrm{CC}}$: Cascaded control for bounding the response times of cloud applications". In: *2018 American Control Conference*. Milwaukee, Wisconsin, USA.

This paper develops models and control strategies for a cloud application hosted on a single server. The goal is to improve the decision making for a graceful degradation concept, denoted as *brownout*, that can lower the response quality when needed in order to control response times. The proposed control strategy utilizes a cascaded structure that is able to improve the predictability of the brownout concept, while providing a model-based control design. The proposed design is compared to previous strategies in a simulation campaign.

The idea to try to improve the brownout control design came from K.-E. Årzén and M. Maggio. The cascaded control structure and models were proposed by T. Nylander after discussions with the co-authors. The simulations were performed by T. Nylander, extending on a simulator previously built by

C. Klein and M. Maggio. The manuscript was written by T. Nylander with many inputs and comments from the co-authors.

Paper II

Berner, T., J. Ruuskanen, M. Maggio, and K.-E. Årzén (2022). "Improved dynamic modeling for controlled server queues". Under journal submission.

This paper builds on the server modeling from Paper I, and identifies certain server characteristics where the previous model is insufficient. Specifically, the involved characteristics are represented by *queuing disciplines*, and it is shown that these affect the server dynamics. A new, nonlinear model structure is proposed based on a simulation study as well as on other known queuing theoretic relations. The proposed structure can represent the dynamics for the more general LPS discipline, resulting in a more versatile model. A control design example showcases the benefits of the structure, using a linearized version of the proposed model. The design example highlights critical frequency ranges where the shape of the involved service time distributions need to be taken into account. Both the proposed model structure and the control design example are evaluated using simulations.

The idea to investigate the effect of the queuing discipline on server dynamics came from T. Berner. The final model structure was developed by T. Berner from discussions with the co-authors. The idea to include a control design example was proposed by K.-E. Årzén. The simulations were performed by T. Berner, extending on the simulator from Paper I. The manuscript was written by T. Berner with many inputs and comments from the co-authors.

Paper III

Nylander, T., M. Thelander Andrén, K.-E. Årzén, and M. Maggio (2018). "Cloud application predictability through integrated load-balancing and service time control". In: *Proceedings of the 15th IEEE International Conference on Autonomic Computing*. Trento, Italy (Received Best Paper Award).

This paper is focused on interactions between different decision making layers in the cloud, for an application hosted on multiple servers. The paper shows that negative interactions can occur when load balancing policies and server controllers are not designed with respect to each other. This is exemplified for the popular load balancing strategy JSQ, that in combination with brownout server controllers can result in a heavy oscillatory behavior. To mitigate this potential problem, an alternative decision making structure is proposed where both layers are co-designed. The proposed design is

evaluated and compared to previous structures in an extensive simulation campaign.

The idea to investigate interactions between decision making layers in the cloud came from T. Nylander. The problem statement, including the highlighting of the JSQ strategy, was then discussed with the other co-authors. The proposed co-design structure was developed by T. Nylander and M. Thelander Andrén, with inputs from the other co-authors as well. The simulations were performed by T. Nylander and M. Thelander Andrén, utilizing an extension of the simulator from Paper I. The manuscript was written by T. Nylander and M. Thelander Andrén, with inputs and comments from the other co-authors as well.

Paper IV

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020). "Modeling of request cloning in cloud server systems using processor sharing". In: *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering.* Edmonton, Canada.

This paper proposes a simplified method to model request cloning in cloud systems by introducing the concept of *synchronized service.* As the method has no assumptions on the statistical distributions involved, it allows for re-using of previous queuing theoretic results, as the model of cloning to n servers becomes equivalent to a basic single server model. The paper presents examples of server systems that fulfill the synchronized service criterion, but also investigates cases where the criterion can not be fulfilled. Specifically, it proves error bounds for common imperfections such as delays in arrival and cancellation of the clones. Additionally, it presents an intuitive reasoning as to why request cloning in a server system involving the popular load balancing policy JSQ approximately behaves as a synchronized system. The proposed cloning model with all examples and imperfections are evaluated using simulations.

The idea to try to model request cloning using the synchronized service criterion came from T. Nylander. The formalization of the concept and the applications presented in the paper were discussed with all co-authors, especially J. Ruuskanen. The idea to include analysis for imperfect systems came from T. Nylander and J. Ruuskanen, and the formalization and proofs for the error bounds were developed by J. Ruuskanen. The simulations were performed by T. Nylander and J. Ruuskanen, using a simulator extending on the one developed in Paper I. The manuscript was written by T. Nylander and J. Ruuskanen, with inputs and comments from the other co-authors as well.

Paper V

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020). “Towards performance modeling of speculative execution for cloud applications”. In: *ACM/SPEC International Conference on Performance Engineering Companion (ICPE ’20 Companion).* Edmonton, Canada.

This paper builds on the modeling principles from Paper IV for request cloning, and extends the scope to model the more general cloning concept of speculative execution. Here the request clones are sent after a given speculation time, and not simultaneously along with the original request as in Paper IV. The modeling in this paper utilizes the near-synchronization property of the JSQ load-balancer, enabling a simplified, yet accurate, approximate analysis for any configuration of speculation times and number of clones. As in Paper IV, there are no assumptions on the properties of the statistical distributions. The paper presents the modeling procedure, together with an illustrative example. The modeling is evaluated using simulations that highlight the key promises of the proposed method.

The idea to utilize the near-synchronization property of JSQ to model speculative execution came from T. Nylander, and was refined in discussions together with the other co-authors. The formalization of the procedure was developed by J.Ruuskanen together with T. Nylander. The simulations were performed by T.Nylander and J. Ruuskanen, using a simulator extending on the one developed in Paper IV. The manuscript was written by T. Nylander and J. Ruuskanen, with inputs and comments from the other co-authors as well.

Additional Publications

The following publication by the author is not included in the thesis:

Ruuskanen, J., T. Berner, K.-E. Årzén, and A. Cervin (2021). “Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing”. *Performance Evaluation* **151**.

2

Background

In this thesis, methods and concepts from both queuing theory and control theory are applied in a cloud computing setting. The analysis is performed on a high abstraction level, i.e., cloud specific tools and programs are mostly left out in the included papers. Instead, more general concepts such as *servers* and high level architectures are utilized. However, in order to provide context and motivation, this background section includes descriptions of the most relevant concepts within cloud computing. Furthermore, the basic building blocks of queuing theory are also presented, as they are utilized in all papers in this thesis, especially in Papers IV - V. Finally, some classic concepts and strategies from control theory that are applied in a cloud setting in Papers I - III, are presented here to aid the reader.

2.1 Cloud Computing

Cloud computing can be described as the illusion of being provided infinite computing resources. However, there exists more precise definitions. The classical definition from National Institute of Standards and Technology (NIST) provides the following essential characteristics [Mell and Grance, 2011]:

- On-demand self-service – The cloud user should be able to automatically acquire computing resources, such as servers and storage.

- Broad network access – The cloud should be accessible through standard protocols that promote use by heterogeneous devices, e.g., mobile phones and laptops.

- Resource pooling – Physical and virtual resources are dynamically assigned to multiple cloud users through resource pooling. The users are not in control of where the computations are located.

Figure 2.1 Server racks in a Facebook data center in Luleå, Sweden. Picture from [Brodkin, 2013].

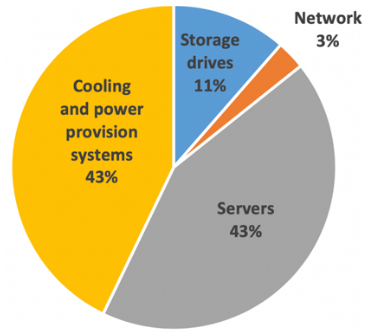


Figure 2.2 Typical power usage share in a data center. Figure from [Shehabi et al., 2016].

- Rapid elasticity – Resources should be elastically provisioned and released, in order to scale both up and down with the demand. This characteristic is what provides the illusion of infinite resources.

- Measured service – Resource usage should be monitored in order to be controlled and optimized. The measures should be reported to ensure transparency for the cloud users.

Together with the essential characteristics, the NIST definition also provides the following cloud deployment models: (i) Private; (ii) Community; (iii) Public; and (iv) Hybrid. Defining the access of resources, the first three are self-explanatory whereas the hybrid model acts as a composition of the other models. In this thesis, the cloud applications are assumed to be hosted on a public cloud, where users from anywhere can interact with the application.

Data Centers

The computations in the cloud are typically performed in large-scale data centers, containing thousands of physical servers together with storage, switches, routers, cables and cooling systems [Barroso and Hölzle, 2009]. The servers are stored in racks as shown in Figure 2.1, and as can be seen in Figure 2.2, the majority of the power is consumed by the servers and cooling systems [Shehabi et al., 2016]. As a large data center in the US consumes several hundreds of megawatt (MW) [U.S. Department Of Energy, 2020], corresponding to about half of the power created by a nuclear reactor, energy efficiency is a key factor in cloud computing. In 2018, the estimated global data center energy usage was 205 TWh, or around 1% of global electricy consumption [Masanet et al., 2020].

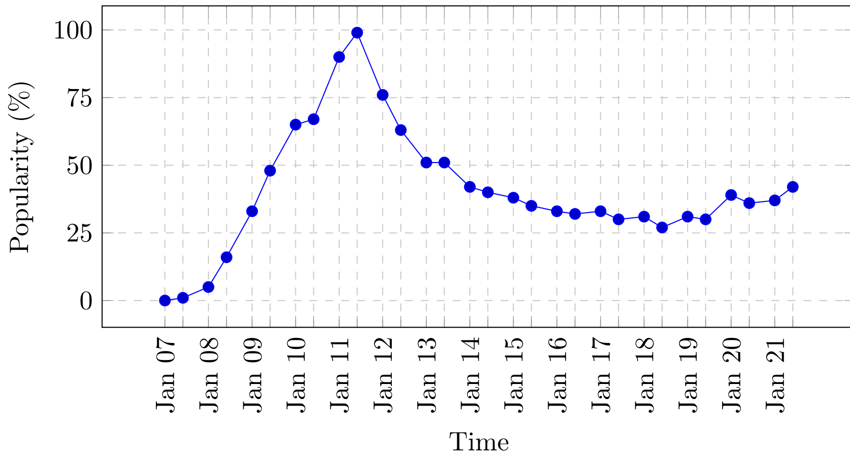


Figure 2.3 Cloud computing search term popularity, according to Google Trends: `https://trends.google.com`. Data collected Dec 2021.

Trends and Research Topics

The term cloud computing was defined by NIST [Mell and Grance, 2011] in 2011, however, the concept started to gain popularity already a couple of years earlier. This can be seen in Figure 2.3, that shows the popularity of the search term "cloud computing" during 2007-2021, with data collected from Google Trends in December 2021. The popularity of the search term peaked in 2011, but as it is on levels of 40-50% of peak value today, cloud computing is still both relevant and important in the everyday life.

The research scope of cloud computing has through the years shown to be quite wide and diverse. As found by the literature review paper [Bayramusta and Nasir, 2016], the five most dominant themes in the years 2009-2014 were:

- Cloud Computing Adoption (19% of all 236 reviewed articles)
- Legal & Ethical Dimension of Cloud Computing (15%)
- Conceptualization & Evolution of Cloud Computing (14%)
- Technical Dimension of Cloud Computing (13%)
- Application of Cloud Computing in Different Sectors (10%)

Among the least popular themes during 2009-2014 were Cloud Computing for Mobile Applications (6%) and Energy Consumption Dimension of Cloud Computing (4%).

A more recent paper that tries to look ahead on the new trends and research directions within cloud [Varghese and Buyya, 2018] highlights the following topics:

- Distributed cloud infrastructure and edge/fog computing
- Multi-tier cloud architectures, i.e., microservices
- Cloud computing impact on both societal and scientific avenues
- Cloud security and sustainability for architecting future systems

It is interesting to compare the two papers as a shift towards the topics that were unpopular in the beginning of the cloud era, i.e., security and sustainability, have become increasingly more popular. Another major trend is the focus towards distributed cloud applications, where the computations are not only processed in large-scale data centers, but also closer to the end-users on edge devices such as network base stations.

Virtualization

The concept of virtualization allows cloud users to share physical resources in cloud data centers, while obtaining an isolated environment [Jain and Choudhary, 2016]. Two common approaches include both full hardware and container-based virtualization.

In the first approach, a virtual machine (VM) is spawned for each user that acts like a physical computer with its own operating system. The VMs are administrated by a hypervisor [Smith and Nair, 2005]. While this approach allows for flexibility (host machine and guest VM can use different operating systems) and enhanced isolation, the full hardware virtualization does create a lot of overhead. The average startup time for a VM in a common public cloud is approximately a couple of minutes [Mao and Humphrey, 2012]. This startup delay makes it difficult to host cloud applications for rapidly changing workloads.

In the second virtualization approach, denoted as containerization or operating-system-level virtualization, a container is spawned for each user. This concept is less flexible, as the guest container needs to use the same operating system as the host machine. However, it does reduce a lot of the overhead present in VMs [Soltesz et al., 2007]. The typical startup time for a container is in the order of a couple of seconds [Medel et al., 2016], which makes them very useful for cloud application deployments.

Service Models

The cloud providers offer computing and services to its users. There exists many different service models, which define what type of product

that the cloud provider makes available. The three original models are (i) Infrastructure-as-a-Service (IaaS); (ii) Platform-as-a-Service (PaaS); and (iii) Software-as-a-Service (SaaS) [Mell and Grance, 2011].

IaaS is the model that provides the least amount of abstraction, thus many administrative tasks need to be performed by the cloud user. This service model provides the users with virtualized hardware, e.g., VMs and storage services, that can be used to build up the complete infrastructure needed to host an application [Bokhari et al., 2016]. The cloud users pay for their usage and are responsible for scaling the application. Examples of IaaS providers include Amazon Web Services (AWS), Microsoft Azure and Google Cloud.

PaaS provides a bit more abstraction as the server management is now included in the offering, which means that the cloud users are not responsible for, e.g., autoscaling of the application. The cloud provider hosts both hardware and software (including operative system) that allows application programmers to build their software on top of the platform [Bokhari et al., 2016]. Examples include AWS Elastic Beanstalk and Google App Engine.

SaaS is targeted towards the end-users as it delivers a complete application hosted in the cloud, and accessible via, e.g., a web browser. Bug fixes and software updates are managed by the provider, and no installation is required for the end-users. This makes these services both easy to use and suitable for collaboration [Bokhari et al., 2016]. Examples of SaaS applications include Netflix, Dropbox and Gmail.

In recent years, the concept of Function-as-a-Service (FaaS), or serverless computing, has also increased in popularity. In this service model, the application programmer is provided with an interface to develop functions that run in the cloud, where the computing nodes are automatically scaled [Jonas et al., 2019]. The most common example of FaaS is AWS Lambda, where the billing of the service is based on every millisecond of computation time.

Elasticity

One of the most important characteristics of cloud computing is, as formalized in the NIST definition [Mell and Grance, 2011], the elasticity of the cloud applications, i.e., the ability to scale. For some cloud service models, such as PaaS and FaaS, the autoscaling is already performed and managed by the cloud provider. However, for the widely used IaaS model, the cloud user is responsible for ensuring the elasticity. For the PaaS model, some platforms include autoscaling abilities, for example AWS Elastic Beanstalk. Another example of a platform that includes autoscaling is Kubernetes, which is an open source platform for managing containerized applications [Burns et al., 2018]. As the code is open source, the user can edit the autoscaling policies to improve the responsiveness of the application, thus studying algorithms and

models for elastic cloud applications can be relevant even for the platform setting.

The autoscaling of cloud applications has been an important research topic in the last decade, with many different algorithms proposed [Lorido-Botran et al., 2014]. The scaling can be performed in two dimensions, both horizontal (adding/removing of servers), as well as vertical (increasing/decreasing speed of servers) [Millnert and Eker, 2020]. Most of the proposed algorithms can be grouped into using one or many of the following techniques: (i) Threshold-based rules; (ii) Reinforcement learning; (iii) Queuing theory; (iv) Control theory; and (v) Time series analysis. Another way of categorizing the autoscaling policies is by their anticipation capabilities, i.e., as reactive or proactive, where the latter includes some sort of prediction of future resource needs [Lorido-Botran et al., 2014].

Microservices

In the recent years, cloud application development has largely shifted from large monolithic applications to instead be composed of independent and loosely coupled microservices [Ueda et al., 2016]. With this architecture, one user-facing application could consist of hundreds, or thousands, of microservices that communicate through remote calls using a common application programming interface (API). All of these microservices can then be deployed, scaled and managed independently.

The modularity of this approach can simplify both deployment and debugging of cloud applications [Gan and Delimitrou, 2018]. As errors in the application can be tracked to an isolated component, it both becomes easier to track down the bug as well as deploying the update for the fix. Moreover, microservices offer flexibility to the applications developers as many different programming languages and frameworks can be used to build up the application. In addition, the microservices architecture fits well with the increasingly popular containerization model for virtualization in the cloud.

However, the distributed manner of a microservices application in the cloud also poses challenges. The development task can get more difficult, as it becomes harder for the teams to see the big picture of the application [Balalaie et al., 2015]. Furthermore, more time is spent on processing and sending network requests, which can deteriorate the performance of the application [Gan et al., 2019]. With this architecture, it also becomes more difficult to develop performance models for application end-to-end response times and to, e.g., find bottlenecks in the system [Ueda et al., 2016].

Graceful Degradation

A common method for keeping tail latencies low in cloud applications is the concept of graceful degradation. For cloud applications under periods of high

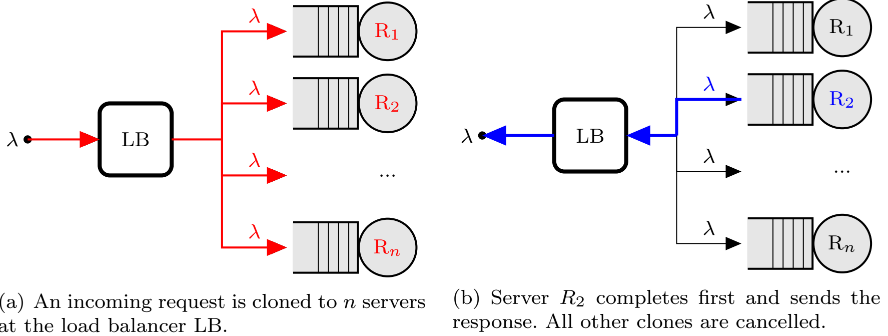
user loads, the application deteriorates its user content to some degree in order to increase the service rate. These techniques are usually only meant to be used for short periods of time, for instance while waiting for more servers to start, as it of course is not desirable to provide the users with degraded application content [Nylander et al., 2018].

The most extreme case of graceful degradation is admission control [Konstanteli et al., 2012], where user requests are denied to enter the application if the servers are too overloaded. This is an effective way of keeping tail latencies low and predictable, however, the consequences of completely denying user service can outweigh its advantages. Another graceful degradation technique is to limit the time user requests can spend in the application, and iteratively refine the response until the maximum time is reached [Ding et al., 2011]. This is, however, only applicable to certain types of requests such as search queries. Finally, another option for implementing graceful degradation behavior is Brownout [Klein et al., 2014]. In this technique the content of the application response is split into two parts, one mandatory and one optional. The mandatory part is composed by the core features of the application response and is always sent, whereas the optional part that includes nice-to-have features is only sent when the application is not under heavy load. All of these mentioned techniques can be utilized as actuators that realize control strategies that, e.g., try to keep tail latencies below some maximum setpoint.

Cloning and Speculative Execution

A common performance issue with large and distributed cloud applications is the occurrence of *stragglers*, i.e., requests that for different reasons require significantly longer time to execute than the rest. The problem is extra severe when other parts of the application depend on the completion of the straggler requests [Dean and Barroso, 2013], and this will especially affect the tail latencies of the application. The causes for these slow requests can be many: software bugs, network issues, hardware failures or inconsistent performance of the virtualized environments. What they share in common is that these causes are difficult to predict in advance, which means that the straggler problem can not always be mitigated by standard load balancing algorithms.

One mitigation technique that has received a lot of research attention in recent years is cloning, where multiple clones of the same original request are sent simultaneously to different servers [Ananthanarayanan et al., 2013]. There exists different variations of the cloning technique, and the two most common are (i) cancel-on-complete (CoC); (ii) cancel-on-start (CoS); and (iii) speculative execution. In CoC cloning, all clones execute until the first one completes, and immediately after the first completion all other clones cancel their execution. This is shown schematically in Figure 2.4. If not oth-



(a) An incoming request is cloned to n servers at the load balancer LB.

(b) Server R_2 completes first and sends the response. All other clones are cancelled.

Figure 2.4 Cancel-on-complete cloning to n servers.

erwise mentioned, the concept cloning in this thesis refers to the CoC variant. The CoS variant cancels all clones as the first one *starts* its execution, which generally leads to a reduced system load compared to the CoC variant. However, as only one clone gets processed, the CoS variant is not as effective on mitigating issues with, e.g., inconsistent VM or container performance. The speculative execution concept is identical to CoC cloning, with the difference that the sending of clones can be delayed by a *speculation time*. As is the case with CoS cloning, speculative execution also leads to less overhead due to the delayed clones.

Cloning has for quite some time been shown to be able to work well in practice, when evaluated in actual large-scale cloud environments [Ananthanarayanan et al., 2013]. The intuition behind why it should be successful at mitigating stragglers and keeping tail latencies low, is that the more servers that serve a specific request, the more likely the request is to find a fast performing server. However, it is then likewise intuitive to assume that the extra clones will increase the system load and therefore make all servers slower. The need for analysis and modeling of the effects of cloning is thus clear, in order to determine under what circumstances cloning increases the application performance, and when it deteriorates performance. The first exact analysis was published in 2015 [Gardner et al., 2015], and its involved statistical assumptions were restricted towards the exponential distribution. Since then, more publications have widened the scope and relaxed the assumptions [Joshi et al., 2015; Qiu et al., 2016] but the concept of cloning still remains as a fairly open research topic.

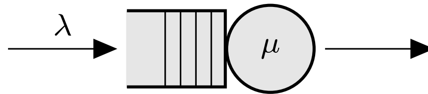


Figure 2.5 Single server model. Requests arrive with incoming arrival rate λ, and are processed with outgoing service rate μ.

2.2 Queuing Theory

In this thesis, concepts from the field of queuing theory are utilized in all Papers I-V. This section aims to give a background relevant to the scope of the thesis, and a complete introduction can be found in the book by Kleinrock [Kleinrock, 1975]. Queuing-theoretical methods can be used to analyze any application that involves queues, e.g., from grocery store checkouts to customer services. In the context of cloud computing and computing systems in general, the queues of interest are within the servers, where requests wait to be processed. With the queuing theory framework, it is possible to analyze computing systems and predict cloud application response times and capacity requirements.

Single Server

The most elementary component that queuing theory can model is the single server, shown in Figure 2.5, that represents a unit capable of performing computation. The modeling relies heavily on statistical distributions, and the stream of arriving requests is described by the inter-arrival time distribution with cumulative distribution function (CDF) $F_{\text{arr}}(x)$ and rate λ. The requests are put in a queue before being processed by the server, described by its service time distribution with CDF $F_{\text{ser}}(x)$ and rate μ. The involved distributions can also be represented by their probability density functions (PDF), and for, e.g., service times the notation becomes $f_{\text{ser}}(x) = \frac{d}{dx}F_{\text{ser}}(x)$. The most common single server model is denoted in Kendall's notation [Kendall, 1953] as $M/M/1$, where M represents the exponential distribution for both inter-arrivals and service times. Other more general model notations include $M/G/1$ (exponential inter-arrivals and general service times) as well as $G/G/1$ (general inter-arrivals and service times).

Queuing Disciplines

The processing of the requests in the server is modeled by queuing disciplines [Kleinrock, 1975]. The three common types that are used in this thesis are shown in Figure 2.6. The simplest possible discipline is first-come first-served (FCFS) shown in Figure 2.6(a), where only one request is simultaneously processed, and the other wait in a queue according to order of arrival. This queuing discipline is an accurate representation for non-preemptible

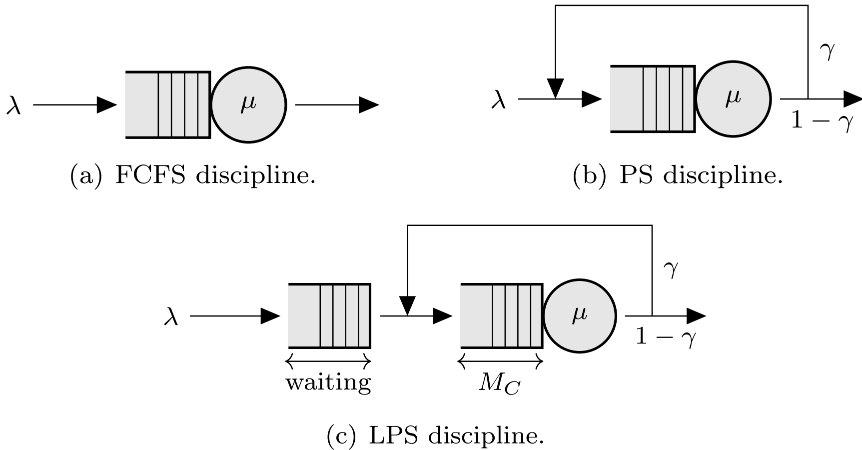
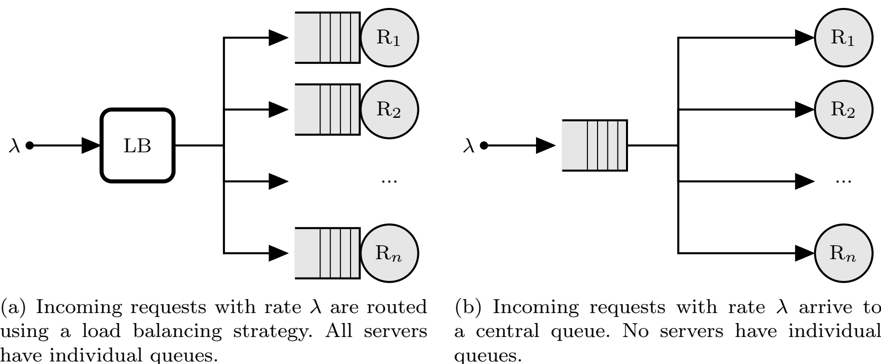


Figure 2.6 Queuing discipline models.

requests, i.e., requests that must run from start to completion without interrupts.

Figure 2.6(b) shows another commonly used discipline denoted as processor sharing (PS). With this concept, each request is executed for a short period of time ϵ before it gets put back last in the queue unless they finish. This is an accurate representation of round-robin scheduling of requests that allow preemption. In Figure 2.6(b), γ represents the share of non-finished requests that get placed in the queue again, whereas $1-\gamma$ is the share of finished requests that leave the queue. As the execution time ϵ approaches zero, the PS discipline can be viewed as all requests simultaneously processing in the server, where each request gets the same share of the current processing capacity, hence the name of the discipline. A drawback with implementing this concept is that for many simultaneous requests, the overhead of the switching between the requests can become large [Zhang et al., 2009].

A possible remedy to the potential switching overhead problem of the PS discipline, is to instead only allow a maximum number of concurrent requests, denoted as M_C, to execute simultaneously. This queuing discipline is denoted as limited processor sharing (LPS), see, e.g., [Zhang et al., 2009], and is shown in Figure 2.6(c). As can be seen, it behaves as a combination of the FCFS and PS disciplines, where a maximum of M_C concurrent requests execute in the server in a PS fashion, and any number of requests beyond M_C wait in a queue in order of arrival. The LPS discipline thus serves as a generalization of the special cases FCFS ($M_C = 1$) and PS ($M_C = \infty$).



(a) Incoming requests with rate λ are routed using a load balancing strategy. All servers have individual queues.

(b) Incoming requests with rate λ arrive to a central queue. No servers have individual queues.

Figure 2.7 Two common server systems.

Server Systems

Replicated cloud applications can be represented by *systems* of servers in queuing theory. Figure 2.7 shows two common structures where one application is replicated over n servers.

Figure 2.7(a) shows a setup where incoming requests with rate λ arrive at a load balancer (LB), that routes the requests to a server with an individual queue. The routing is performed according to some load balancing strategy [Sharma et al., 2008]. Two of the most common strategies include

- Random [Sharma et al., 2008] – The simplest possible strategy, where servers are chosen at random. Has minimal decision-making delays and simplifies stability analysis, but is in general far from optimal.
- Join-Shortest-Queue (JSQ) [Gupta et al., 2007] – Routes to server with least requests, i.e., the shortest queue. Requires measurements of all server queue lengths, but achieves near-optimal performance.

Many other load balancing algorithms exist, see [Ghomi et al., 2017] for an overview of algorithms implemented in some popular cloud platforms.

Figure 2.7(b) shows a different setup where no load balancing strategy is necessary as all requests instead wait in a central queue in a FCFS fashion [Sharma et al., 2008]. Each server only processes one request at a time, and upon the event of a finished request the server is provided a new request from the front of the central queue. This setup has a model notation similar to the single server case, for general distributions and n servers its notation becomes $G/G/n$. This structure makes the server routing trivial, however, the single queue can become a bottleneck if the request streams are large. Also, this setup does not allow requests to share servers which can limit the processing efficiency for some request types.

More complicated server systems exist as well, that describe how different queues interact in a network [Chandy et al., 1975]. However, these queuing networks are out of scope for this thesis, as they are not required for the analysis performed in the papers included.

Performance Modeling

The traditional models in queuing theory consider stationary relationships, as transient and dynamic models generally become difficult to find closed-form expressions for [Kleinrock, 1975]. Stationary metrics of interest include stability, i.e., if the queue lengths stay bounded, utilization $0 \leq \rho = \lambda/\mu \leq 1$ and average response time. A very useful stationary relationship that holds in any type of queuing system is *Little's Law* [Little, 1961]:

$$E(N) = \bar{\lambda} E(T), \tag{2.1}$$

which states that the average number of requests $E(N)$ in a system is equal to the effective arrival rate $\bar{\lambda}$, defined by the rate of requests that enter the queue, multiplied by the average time $E(T)$ a request spends in the system. In any queuing system, if either $E(N)$ or $E(T)$ is found, the other metric can be obtained through Equation (2.1).

For the single server $M/M/1$ case it is possible to obtain many performance metrics explicitly. $\lambda < \mu$ is required for stability and the average response time can through Little's Law be determined as

$$E(T) = \frac{1}{\mu - \lambda}, \tag{2.2}$$

which holds for both FCFS and PS queuing disciplines. For the more general $M/G/1$ case, the expression for average response time for the FCFS case can be found through the Pollaczek-Khinchin mean formula (see, e.g., [Kleinrock, 1975]) as

$$E(T) = E(x) + \frac{\rho E(x)(1 + C_s^2)}{2(1 - \rho)}, \tag{2.3}$$

where $E(x)$ is the average service time and C_s^2 is the squared coefficient of variation of the service time distribution. For the PS case, the expression only depends on the average service time $E(x)$ and arrival rate lambda:

$$E(T) = \frac{E(x)}{1 - \lambda E(x)}. \tag{2.4}$$

Stationary performance models also exist for the two server systems shown in Figure 2.7. For the random load balancer, the properties of the inter-arrival times are preserved and all single server results can thus be used for each server separately. For the JSQ strategy no exact performance

metrics exist, but an accurate expression for the average response time, assuming the PS queuing discipline, is given by [Gupta et al., 2007]. It can not be stated as a simple closed-form expression, however, it approximates the average response time of the JSQ server system with n servers with an error within 2-3% using only the arrival rate λ and the average service time $E(x)$. For the central queue case, exact expressions for average response time only exist for the $M/M/n$ model, i.e., assuming exponential distributions. As the expression is complicated it is left out for brevity, but can be found in, e.g., [Kleinrock, 1975]. For the more general $M/G/n$ case, no exact performance metrics exist. Furthermore, it is shown in [Gupta et al., 2010] that the first two moments of the service time distribution are unfortunately not enough to provide an approximation that is accurate under all loads.

Dynamic performance models are more interesting from a control-theoretical point of view. For the remaining work w in a server, the dynamics are given by the following well-known result (see, e.g., [Jean-Marie and Robert, 1994]), that holds for any work-conserving queuing discipline that never intentionally keep servers idle, including FCFS, LPS and PS:

$$\dot{w} = \lambda\, E(x) - 1. \tag{2.5}$$

For the queue length growth process, i.e., when $\lambda > \mu$, there exists exact dynamical models for both FCFS and PS [Jean-Marie and Robert, 1994]. For the FCFS case, the model is linear, simple and depends only on the arrival rate λ and service rate μ as

$$\dot{q} = \lambda - \mu, \tag{2.6}$$

with queue length q. However, for the PS case the model instead becomes the following asymptotical model

$$\lim_{t\to\infty} \dot{q} = \alpha, \tag{2.7}$$

where the asymptotical growth rate $\alpha > 0$ is the solution to the integral equation

$$\alpha = \lambda \left(1 - \int_0^\infty e^{-\alpha x} f_{ser}\,(x)\, dx\right), \tag{2.8}$$

with $f_{ser}(x)$ the PDF of the service time distribution. As can be seen in Equation (2.8), the growth rate α depends on the *entire* service time distribution for PS, and not only its mean.

For the case of stable non-stationary queues, i.e., with $\lambda < \mu$, there only exists approximations for the queue length dynamics. One well known is the pointwise stationary fluid flow approximation (PSFFA) [Wang et al., 1996], that includes support for many different service time distributions. The model

is fluid, i.e., it does not consider the individual requests in the queue, rather it views the requests as a *flow* and the derivative of the queue is determined as $\dot{q}$ = flow in − flow out. For the well-known exponential case, sometimes denoted as Tipper's model, the equation becomes

$$\dot{q} = \lambda - \mu \left(\frac{q}{q+1} \right). \tag{2.9}$$

Distribution of the Minimum

An old statistical result relevant for queuing-theoretic applications, when modeling request cloning, is presented here. The concept concerns a statistical case where, for each random draw of n random variables, the minimum value is always chosen. The result is formalized in the following theorem:

THEOREM 1
(Distribution of the Minimum) Given a set of n random variables $\{X_1, \dots, X_n\}$ with **any** CDF, and denoting with $F_i(x)$ the CDF of X_i; the CDF of the random variable $X_{\min}$, where $X_{\min} = \min\{X_1, \dots, X_n\}$ becomes

$$\begin{aligned} F_{\min}(x) \quad = \quad & (-1)^0 \sum_{i=1}^{n} F_i(x) + \\ & (-1)^1 \sum_{i<j} F_{i,j}(x,x) + \\ & (-1)^2 \sum_{i<j<k} F_{i,j,k}(x,x,x) + \dots + \\ & (-1)^{n-1} F_{i,j,\dots,n}(x, \cdots, x), \end{aligned} \tag{2.10}$$

where $F_{i,j}(x,x)$ is the joint CDF of random variables X_i and X_j. If X_i and X_j are independent, i.e., if $F_{i,j}(x,x) = F_i(x)F_j(x)$, Equation (2.10) becomes

$$F_{\min}(x) = 1 - \prod_{i=1}^{n} \{1 - F_i(x)\}. \tag{2.11}$$

Proof. See Theorem 1 in Paper IV. □

This theorem is very useful, as it provides a method for calculating the *complete* statistical distribution $F_{\min}(x)$ for the choice of the minimum value among n random variables. Additionally, in Equation (2.10) there are *no* assumptions on the involved distributions, which makes the theorem powerful. Finally, if assumptions on independent variables are used, the algorithm for $F_{\min}(x)$ becomes the much simpler expression given in Equation (2.11).

2.3 Control Concepts

This section presents some relevant concepts within control theory, that are utilized in a cloud setting in this thesis. The reader is assumed to possess some basic knowledge of the subject of automatic control. For a complete introduction to the topic, see [Åström and Murray, 2019].

Sensors and Actuators

The main purpose of applied control theory is to affect the system of interest, such that it behaves in a desirable way. As the applications of control theory are widespread, the system under control could be anything from a water tank to an aircraft autopilot. One property that these diverse systems share, is that in order to successfully control them, efficient sensors and actuators are needed [Åström and Murray, 2019]. In many classical control applications, these are physical and not of particular interest. However, in the computing systems domain, both sensors and actuators can be *designed* in order to simplify modeling and control, as these can be part of the software itself [Maggio et al., 2012].

Modeling

A basic prerequisite for applying control-theoretical methods is the ability to construct models useful for control. The models describe how control inputs u and the state x affects the dynamics of the output y, as exemplified in the following linear equation:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du, \end{aligned} \tag{2.12}$$

with A a square matrix, B, C and D vectors, x a vector and u and y scalars. The model (2.12) is an example of a linear single-input single-output (SISO) system on state-space form, however, the control-theoretical models could of course be both non-linear and have multiple inputs and outputs (MIMO) [Ljung, 2000]. In this thesis, both linear and non-linear SISO models will be utilized.

If the model is linear, then it can, through the Laplace transform (see, e.g., [Widder, 2015]), be stated as a transfer function [Åström and Murray, 2019] as:

$$Y(s) = \Big(C\left(sI - A\right)^{-1} B + D\Big)\, U(s) = G(s)U(s), \tag{2.13}$$

with G the transfer function from Laplace variables U to Y. The *poles* and *zeros* of $G(s)$ can be used to represent important properties of the model, such as stability and speed.

The models in control theory are generally quite coarse, as in order to design a successful controller, only the *relevant* dynamics need to be captured.

If for example a process is to be controlled slowly, then any considerably faster stable poles and zeros can be neglected in the control design [Åström and Murray, 2019].

Feedback and Feedforward

The core part of applied control theory lies in the feedback loop. In order to ensure that a system behaves desirably, measurements and feedback are required. An example feedback loop is shown in Figure 2.8. This representation of the interconnections between the system components is in the Laplace domain, and describes how a process $P(s)$ is affected by control inputs u and disturbances d, with the goal of measurement y following the setpoint (or reference r). In this figure, the control signal u is composed of both a feedback u_{fb} and a feedforward u_{ff} part.

The feedback part is determined by the feedback controller $C(s)$, that calculates a control signal with respect to the control error e, defined as the difference between the desired setpoint r and process measurement y. The most widely utilized controller structure is the simple, yet effective, PID controller [Åström and Hägglund, 2006]. It is used in control applications ranging from process control in factories [Craig et al., 2011] to computing systems [Leva, 2018]. The PID controller is composed of three parts, a proportional (P), integral (I) and derivative (D) part. A common representation of the complete control algorithm, including a filter on the derivative part, can be stated as [Åström and Hägglund, 2006]:

$$U(s) = K\left(1 + \frac{1}{sT_i} + \frac{sT_d}{1 + sT_d/N}\right)E(s), \tag{2.14}$$

with K proportional gain, T_i integral time constant, T_d derivative time constant and N filter constant. With a correct parameter tuning, the PID controller can be utilized to eliminate stationary errors through its I-part, and to predict future errors through the D-part. With these properties, the PID controller performs well enough in many different applications.

The feedforward part of the control signal in Figure 2.8 is determined by the feedforward controller $F_F(s)$. It does not utilize feedback, but instead it relies solely on models which thus have to be accurate. It can take both the setpoint r and measurable disturbances d as input, depending on the purpose of the controller. For improved setpoint following, the setpoint r is fed to the controller that then utilizes a model from r to y to determine a feedforward control signal. Often this requires model inversion, which means that the feedforward relation often is approximated by a static gain instead [Wittenmark et al., 2002]. For improved disturbance rejection, the measurable disturbance d is instead fed into the controller, which also here utilizes an inverse relation between d and y to attenuate the disturbance before it affects the control

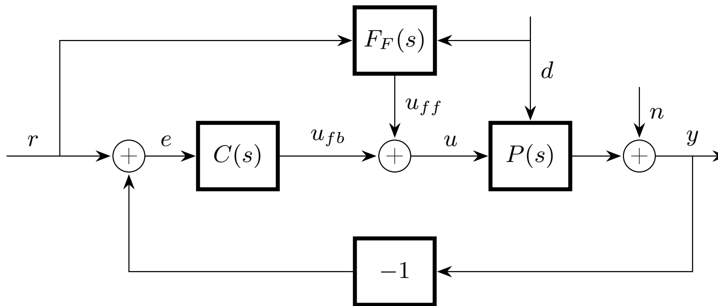


Figure 2.8 Example block diagram for a feedback loop with feedforward. The process $P(s)$ is affected by a measurable disturbance d and measurement noise n. Feedback controller $C(s)$ reacts to changes in error e and feedforward controller to changes in reference r and disturbance d.

loop performance. Feedforward controllers are generally used together with feedback controllers, such as in Figure 2.8.

Linearization Methods

Linearization is an important tool within the field of control theory, as many models of real processes contain nonlinearities, whereas a big part of the theory is based on assumptions of linearity. One common method of linearization is around an operating point x_0, using Taylor series expansions (see, e.g., [Åström and Murray, 2019]). The approximation using this method is then valid close to x_0. The algorithm can be mathematically described as a linearization of a general nonlinear system

$$\begin{aligned} \dot{x} &= f(x,u). \\ y &= g(x,u), \end{aligned} \tag{2.15}$$

with functions f and g. By choosing new variables $\Delta x = x - x_0$, $\Delta u = u - u_0$ and $\Delta y = y - y_0$ the system can be linearized around the stationary point (x_0, u_0, y_0) as

$$\begin{aligned} \Delta\dot{x} &= \tfrac{\partial}{\partial x} f\left(x_0,u_0\right)\Delta x + \tfrac{\partial}{\partial u} f\left(x_0,u_0\right)\Delta u = A\Delta x + B\Delta u \\ \Delta y &= \tfrac{\partial}{\partial x} g\left(x_0,u_0\right)\Delta x + \tfrac{\partial}{\partial u} g\left(x_0,u_0\right)\Delta u = C\Delta x + D\Delta u, \end{aligned} \tag{2.16}$$

i.e., with the same structure as in Equation (2.12).

This method of linearization is accurate enough in many cases, especially if the controlled variables stay close to one operating point at all times. If not, it is possible to extend this method by the concept of *gain scheduling* (see, e.g., [Rugh and Shamma, 2000]). In this control concept, the nonlinear dynamics are linearized according to the algorithm in Equation (2.16) at

multiple operating points, and the controller parameters are calculated for each case. Then, as the system moves between the different operating points, the controller parameters are changed, or scheduled, accordingly. In the extreme case, it is also possible to continuously linearize and adapt the control parameters. A drawback is that for these linearizing methods, the stability can only be guaranteed for slow-varying systems, and explicit performance guarantees might be difficult to prove [Rugh and Shamma, 2000].

Another method of linearization is the *feedback linearization* concept (see, e.g., [Ljung, 2000]). Assume that the nonlinear system is of the following form:

$$\begin{aligned} \dot{x} &= f(x) + ug(x) \\ y &= h(x), \end{aligned} \tag{2.17}$$

where f, g and h are infinitely differentiable functions. The goal with feedback linearization is then to find a control law

$$u = a(x) + vb(x), \tag{2.18}$$

such that the previous system in Equation (2.17) is linear from the new variable v to y. If it is possible to find this relation, then controllers can be designed for v using standard linear control theory, and this linearization is then *exact*.

The concept becomes clearer in this simple sinusoidal example from [Ljung, 2000], with the nonlinear system model

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= x_3 \\ \dot{x}_3 &= u\cos(\arcsin(x_3)) \\ y &= x_1. \end{aligned} \tag{2.19}$$

The feedback control law

$$\Psi: \quad u = (v - x_1 - 3x_2 - 3x_3)/\cos(\arcsin(x_3)) \tag{2.20}$$

brings (2.19) to a linear system with all poles in -1. The dynamics can then further be changed by defining a linear control law for v, i.e., closing the loop from v to y. A block diagram of the example is shown in Figure 2.9, where the inner loop represents the feedback linearization algorithm.

The method of feedback linearization has some clear benefits, as it simplifies the control design in the new variable v. It has been applied in several different settings, ranging from motor control [Chiasson, 1998] to chemical processes [Braake et al., 1998], however, the method does have drawbacks as well. First, it is not applicable to all types of nonlinear systems as they have to be on the same form as Equation (2.17). Second, the feedback law that performs the linearization, i.e., Equation (2.18), in general requires that

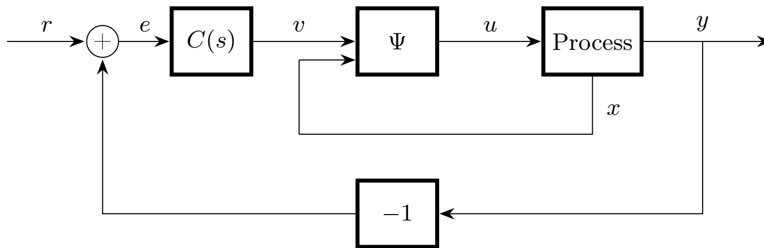


Figure 2.9 Feedback linearization of nonlinear Process defined by Equations (2.19) with linearizing feedback law Ψ from Equation (2.20). Linear controller $C(s)$ can then be designed for the linearized process from v to y.

feedback can be performed on *all* states x. If not all states are measurable, nonlinear observers must be implemented. Finally, if the true system deviates from the model the feedback law does not result in an exact linearization, thus the system's sensitivity towards parameter variations can not be analyzed with simple linear methods.

Cascaded Control

The cascaded control structure in, e.g., [Bolton, 2021], is an important example of a structure that deviates from the basic feedback loop shown in Figure 2.8. This structure, as shown in Figure 2.10, has two loops in cascade and utilizes feedback from *two* measurement signals y_i and y_o. The structure is thus useful when multiple measurements are available, while only one actuator can be used to affect the process. The inner loop controller $C_I(s)$ is the one that actuates on the process through its control signal u, whereas the outer loop controller $C_O(s)$ determines a setpoint r_i for the inner loop.

Using the cascaded structure has several advantages: (i) As the inner loop controller C_I can be designed to be fast, it improves rejection of disturbances d_i acting on the inner process P_I; (ii) Nonlinearities in the inner loop can be linearized using feedback as described in the previous section and exemplified in Figure 2.9; and (iii) The structure introduces a separation of timescales between the two controllers C_I (typically fast) and C_O (typically slower), which simplifies the outer loop control design. The drawback of the structure is that it requires multiple measurement signals to be available, which is not always the case in real-life applications. Also, many simpler processes can be efficiently controlled using a single controller, and in those cases it is not preferable to extend the structure to a cascaded one.

Cascaded control has traditionally been utilized mainly in the chemical processing industry [Wolff and Skogestad, 1996; Alvarez-Ramirez et al., 2002], where one specific example is to closing of an inner loop around valves [Hall, 2017]. The applications have spread to other domains as well, e.g., servo

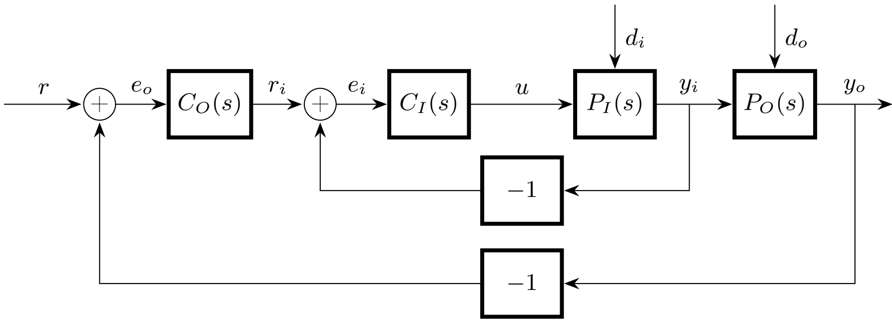


Figure 2.10 A general cascaded feedback structure with inner loop controller $C_I(s)$ and outer loop controller $C_O(s)$ that sets inner setpoint r_i.

systems in automotive industry [Saleem et al., 2015] and robotics [Guo et al., 2008].

Computing System Applications

In the last decades, control theory has been applied more frequently in the computing systems domain. Resource allocation is an area that can be formulated as a control problem, which allows for control-theoretical methods, see, e.g., [Abdelzaher et al., 2002; Hellerstein et al., 2004; Kjaer et al., 2009]. The PID controller is a versatile and simple control structure that is utilized in the computing systems domain as well, [Leva, 2018], where examples include PID control of server queues, clock synchronization in networks and batch data processing. Other examples of applications in the cloud domain include dynamic control of virtual network functions under end-to-end deadline constraints [Millnert et al., 2017], load balancing strategies [Dürango et al., 2014] and event-triggered model predictive control for cluster reconfiguration in data intensive cloud services [Cerf et al., 2016].

Two important dynamic queue length models exist in the control domain, one for the FCFS queuing discipline and one for PS. The FCFS model (see, e.g., [Arcelli et al., 2015]) has very simple dynamics, given by a pure integrator:

$$\dot{q} = \lambda - \frac{1}{\bar{x}}u, \tag{2.21}$$

where $\bar{x}$ is the mean service time and u an actuator that affects the server. The dynamic model for the PS discipline is more complicated and was originally designed for network applications, describing a proportional bandwidth allocation protocol [Paganini et al., 2012]. However, this application is almost completely analogous to the PS queue dynamics, and by minor parameter

adjustments the model becomes the following partial differential equation:

$$\frac{\partial q\left(t,x\right)}{\partial t}=\frac{\partial q\left(t,x\right)}{\partial x}\frac{u\left(t\right)}{q\left(t\right)}+\lambda\,\bar{F}_{ser}\left(x\right), \tag{2.22}$$

with $\bar{F}_{ser}\left(x\right)$ the complementary CDF of service times, i.e., $\bar{F}_{ser}\left(x\right)=1-F_{ser}(x)$.

Both of these models (2.21) and (2.22) are exact in a mean flow model sense, i.e., if multiple queue length evaluations are performed for some sequence of control input u, the mean queue length dynamics will follow the models exactly. The models can be compared to their queuing theoretic counterparts, i.e., the queue length growth models presented earlier in this chapter, for FCFS given in Equation (2.6) and for PS in (2.8). They share the same properties and notably, both FCFS models only depend on the mean of the service time distribution whereas both PS models depend on the *entire* distribution, here represented by the complementary CDF denoted as $\bar{F}_{ser}\left(x\right)$. An important distinction between the control theoretical queue models presented here and queuing theoretic growth models from Section 2.2 is that the control theoretical models have support for *actuators* that can affect the server efficiency, here represented by the control signal u. As a result, the control-theoretical models do not have to assume any relations between arrival rate λ and service rate μ, however, they do imply an indirect assumption on *non-empty* queues, in order for the actuation of control input u to make sense.

3

Improving Cloud Application Predictability

This chapter describes the main contributions of the thesis. The content is mainly divided into two method tracks, where the first utilizes control theory (consisting of Papers I - III), and the second makes use of queuing theory (Papers IV - V). While being different in the mathematical tools used, both tracks share some high-level themes as described in Section 3.4, and also the common goal to improve cloud application predictability. This goal is throughout the thesis interpreted as keeping application tail response times low and consistent, even under uncertainties and sudden events.

The setting assumed in this thesis is from the cloud user point-of-view, i.e., companies or individuals that rent compute capacity in the cloud to host end-user facing applications utilizing an IaaS service model. Both tracks use novel methods and interact with the cloud applications through *actuators*, where the control-theoretical track mainly considers graceful degradation, while the queuing-theoretic methods utilize request cloning.

3.1 Simulation Environment

All papers in this thesis are evaluated in a simulated cloud environment, based on the discrete-event simulator first developed by the authors in [Dürango et al., 2014]. The original simulator included support for configuring (i) inter-arrival distribution for incoming requests; (ii) load balancing strategy and number of server replicas; (iii) service time distributions; and (iv) control strategies in the servers. For the papers included in this thesis, the simulator has been extended and adapted to also include configuration of (i) queuing disciplines: FCFS, PS and LPS; and (ii) request cloning related settings.

The simulator engine is discrete-event based, implemented as an event queue where events are added, changed or removed as the simulation time

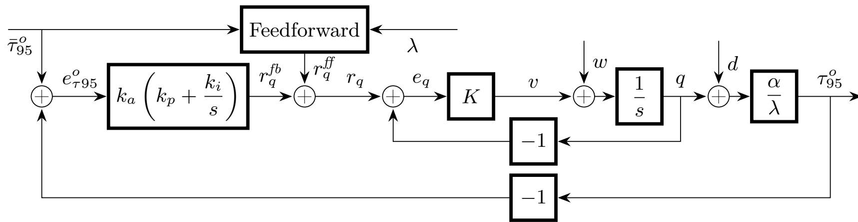


Figure 3.1 The cascaded structure utilized in Paper I, with the proposed models and controllers for both outer and inner loop. Figure from Paper I.

progresses. The evolution of a request consists of the following stages: (i) client module produces a request event; (ii) request arrives at load balancer that decides which server it should be sent to; (iii) request is processed at the chosen server; and (iv) an event notifies request completion and stores metrics such as end-to-end response time. As the simulator is discrete-event based, all control strategies need to be implemented using discretization methods, e.g., as proposed in [Åström and Wittenmark, 1997]. Also, the simulation does not rely on any dynamical (flow) models, as it is implemented in a request-by-request fashion. This makes the simulator a suitable environment for evaluating the dynamical models and strategies proposed in thesis.

3.2 Control-Theoretical Methods

The first method track is focused on applying control-theoretical concepts using mainly graceful degradation as an actuator. The cloud applications considered are replicated over multiple servers, and could represent either a complete (smaller) application or a single microservice that is part of a larger application.

Server Modeling and Control

Papers I and II discuss response time and queue length control for the single server case. The first main contribution to this topic is the proposed cascaded modeling and control structure from Paper I, as seen in Figure 3.1. The inner loop models and controls queue length, a metric that generally is straightforward to measure in a real cloud environment. The actuator that affects the service rate is the graceful degradation concept Brownout, with a binary choice of providing optional content or not. Choosing this binary choice as control signal would make modeling difficult from a control-theoretical perspective. Instead, the approach in Paper I utilizes feedback linearization to transform the system into a control signal denoted v, which represents the derivative of the queue length. In this fashion, the modeling effort and con-

trol design for the inner loop are greatly simplified. The model from v to q becomes an *exact* integrator which can be efficiently controlled by a simple P-controller.

The actuation of control signal v must, however, also be realizable. For this purpose, the binary properties of the Brownout actuator concept are utilized. For each sampling period of length h seconds, a queue length threshold that represents the desired queue length derivative v is calculated. During a sampling period, if the queue length becomes larger than the threshold only mandatory content is served, and if the queue is below, optional content is served as well. In this way, a guaranteed stable actuation of v is achieved, but as a trade off stationary actuation errors are introduced.

These stationary actuation errors can, however, easily be eliminated by the outer loop PI controller as shown in Figure 3.1. Also, as the inner loop is designed to be fast, the design of the outer loop can be made slower to introduce a separation of time scales. Lastly, the design in Paper I also features a feedforward part in the outer loop, that makes the controller react faster to changes in arrival rate λ.

In total, the cascaded structure together with the feedback linearization strategy of the inner loop result in a control design that outperforms the previous control-theoretical efforts made for the Brownout concept [Klein et al., 2014; Maggio et al., 2014].

The second main contribution for the single server case is the improved dynamic queue length modeling proposed in Paper II. Here, a more general continuous graceful degradation actuator is assumed, exemplified by server speed u. As the feedback linearization method for the queue length modeling in Paper I was enabled by the trivial queue length derivative actuation due to the binary property of the Brownout concept, it can be assumed that a different modeling approach is necessary for a more general actuator. Additionally, the results of the evaluation seen in Figures 6-11 in Paper I show that the control strategies behave differently depending on the queuing discipline. These observations acted as inspiration for Paper II, where the key contribution is a model structure that can capture the behavior of a wider range of queuing disciplines.

The foundation of the proposed model structure in Paper II is a simulation-based investigation of the queue length dynamics for disciplines FCFS and PS. The results for FCFS, available in Figure 1 in Paper II, follow the traditional integrator behavior perfectly, and the dynamics are only dependent on the mean value of the service time. For the PS discipline the results are, however, more interesting. As can be seen in Figure 3.2, the behaviors are completely different for the three service time distributions, despite their identical mean values. Another interesting observation that can be made is that the dynamics for the exponential distribution, shown in green in Figure 3.2, follow the pure integrator behavior just as all distributions do

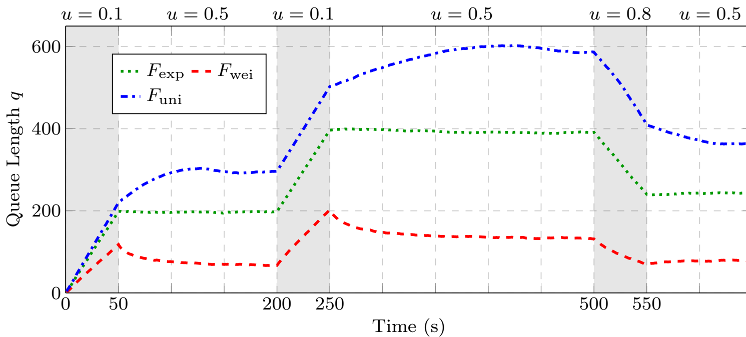


Figure 3.2 Simulation study results for the PS queuing discipline. Service time distributions of types exponential, uniform and Weibull with identical mean values are compared. Figure from Paper II.

for FCFS. A dynamic queue length model for PS must thus include a distribution describing parameter, that for some specific parameter value reduces the dynamics to a pure integrator. Motivated by this insight, and by the fundamental physics involved in the queue, the following novel, nonlinear model structure is proposed:

$$\begin{aligned} \dot{w} &= \lambda \bar{x} - u \\ \dot{q} &= \lambda - k \frac{q}{w} u, \end{aligned} \tag{3.1}$$

with arrival rate λ, queue length q, mean service time $\bar{x}$, remaining work $w > 0$ and distribution specific parameter $k > 0$. The linearization of the model in (3.1) reveals some interesting properties related to k. It is performed around an operating point (w_0, q_o, u_0), which introduces the new variables $\Delta w = w - w_0$, $\Delta q = q - q_0$ and $\Delta u = u - u_0$. The resulting transfer function from ΔU to ΔY then becomes:

$$G_N(s) = -\frac{q_0}{w_0} \frac{\left(1 + s\frac{w_0}{u_0}\right)}{s\left(1 + s\frac{w_0}{ku_0}\right)}. \tag{3.2}$$

As can be seen in Equation (3.2), in addition to an integrator pole, the model also contains a stable pole located at $p_1 = \frac{-ku_0}{w_0}$ and a stable zero at $z_1 = \frac{-u_0}{w_0}$. Clearly, the location of the pole relative to the zero is dependent on the value of the distribution parameter k. For $k < 1$, the pole p_1 is slower than zero z_1, resulting in a phase drop between p_1 and z_1. For $k > 1$, the situation is the opposite and a phase increase is obtained instead. For $k = 1$, the pole and zero cancel out and the linearized model (3.2) becomes a pure integrator.

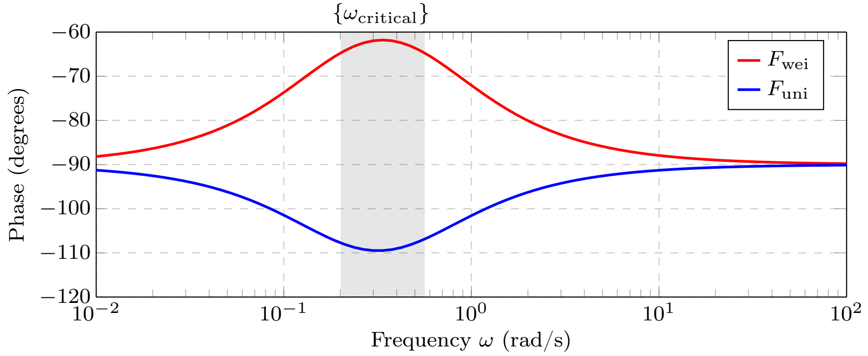


Figure 3.3 Phase part of an example Bode diagram of $G_N(s)$ for distributions of types Weibull and Uniform with identical mean values. $\{\omega_{\text{critical}}\}$ denotes the frequency region between p_1 and z_1. Figure from Paper II.

From a control design perspective, even the distributions with $k \neq 1$ behave integrator-like outside of the frequency region between p_1 and z_1, defined as $\{\omega_{\text{critical}}\}$. This is illustrated in Figure 3.3, that shows the Bode phase of $G_N(s)$ for two example service time distributions of types Weibull (with $k > 1$) and Uniform ($k < 1$). Away from this critical frequency region, i.e., if the control design cut-off frequency is considerably slower or faster than $\{\omega_{\text{critical}}\}$, the controllers can safely be designed using a simple, pure integrator model as the basis, for any distribution. However, if the cut-off frequency is desired to be close to, or inside, $\{\omega_{\text{critical}}\}$, then the distribution specific properties will affect the design. As a phase increase is desirable, distributions with $k > 1$ are easier to design controllers for, and extra effort has to be put into the design for distributions, such as Uniform, with $k < 1$.

The proposed model structure shown in Equation (3.1) was designed with respect to the PS queuing discipline. However, the distribution specific parameter k reduces the model to a pure integrator for $k = 1$, used to represent the Exponential distribution. Thus the model can easily be utilized, through its k-value, to describe the queue dynamics for the more general LPS queuing discipline with any concurrency value M_C. For M_C values close to PS behavior, i.e., if M_C is large, the k-value for a specific distribution will stay close to its PS value. As the M_C value gets closer to 1 (i.e., FCFS-like behavior), the value of k will tend closer and closer to $k = 1$ to represent the changes in behavior. The model structure (3.1) is thus able to represent the full spectrum of LPS behaviors for any M_C, just by altering the value of k.

The proposed model structure is useful both for providing control design related insights, as well as to describe the behaviors of the LPS queuing discipline. However, the model does have drawbacks. The model is only exact

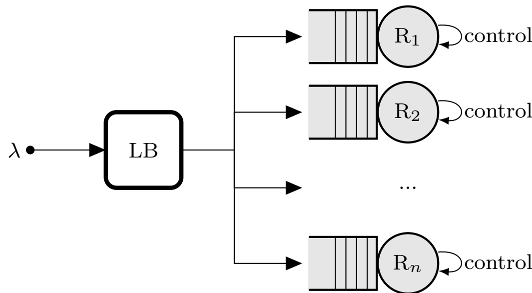


Figure 3.4 The standard load balancing architecture. The load balancer routes incoming requests to a replica, where the request might spend time queuing before service. Replicas include graceful degradation controllers that can interact with the load balancing strategy. Figure from Paper III.

for the FCFS case (i.e., $M_C = 1$), where it reduces to an integrator model of the same form as Equation (2.21) in Section 2.3. For cases where the server concurrency $M_C > 1$, the distribution specific dynamics can not be exactly described by a single parameter k. A comparison can be made towards the exact dynamic model for PS queues, given in Equation (2.22), where the dynamics are dependent on the entire service time distribution. As a result, the value of k has slight variations when, e.g., the incoming arrival rate changes. However, the evaluations show that these variations are generally small and can be tracked well by an online estimation scheme.

Cloud Interactions

Paper III is focused on cloud applications that are replicated over servers, thus requiring a load balancing strategy for routing decisions. As in Papers I and II, the servers still include graceful degradation controllers with the purpose of keeping response times low and predictable. The main contribution for this replicated server architecture is the highlighting of possibly negative interactions that can occur between the load balancing strategy and the graceful degradation controllers in the servers.

Figure 3.4 shows the basic architecture assumed, where the incoming requests get routed to a server replica based on the decisions made by the load balancer, and then processed at the server under the actuation of a graceful degradation controller. In this setup, the load balancer can measure any metrics from the server, but it does not know any of the future decisions to be made by the server controllers. In addition, depending on the queuing discipline implemented in the servers there might be a long delay, caused by server queuing, between the decision made at the load balancer and the control decisions in the servers. This makes the design of the load balancing

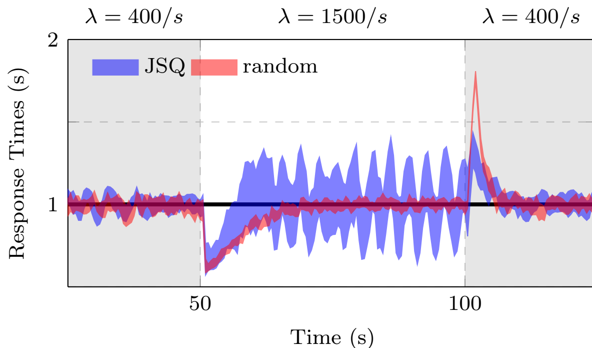


Figure 3.5 Comparison between random and JSQ load balancing. The plot shows setpoint (at 1s) and 95% confidence intervals for the 95$^{\text{th}}$ percentile of response times. Figure from Paper III.

strategy difficult, as the decisions on the different levels of the architecture might be in conflict with each other.

Figure 3.5 shows simulation results of load balancing strategies random and JSQ, when combined with servers subject to cascaded graceful degradation controllers from Paper I in a setup as described in Figure 3.4. The simulations were performed for $n = 5$ servers using the LPS discipline with $M_C = 15$, representing a behavior in between FCFS and PS. The simulations were repeated 20 times in order to construct 95% confidence intervals, and for each run the arrival rate λ shifted between $400/s$ and $1500/s$. The desired setpoint for the 95th percentile of response times is 1 second, and as can be seen the confidence interval for random stays much closer to the setpoint than for JSQ, thus achieving a better predictability. The results for JSQ, on the other hand, exhibit an oscillatory behavior that deteriorates performance, and the oscillations are especially prominent when the arrival rate is $1500/s$. The results are surprising as JSQ widely outperforms random when it comes to load balancing with servers subject to no graceful degradation controllers, as described in Section 2.2.

The results can, however, be intuitively explained if the JSQ load balancing strategy is instead viewed as a feedback queue length controller. The algorithm first collects measurements of all server queue lengths, and then chooses the one with the shortest queue. Using this behavior, JSQ drives all n servers towards the same queue length at each time instant t_k. Thus, JSQ can be interpreted as a feedback controller with an alternating queue length setpoint $r(t_k)$ as

$$r(t_k) = \frac{1}{n} \sum_{i=1}^{n} q_i(t_k). \tag{3.3}$$

JSQ does not have an integrator-like behavior, as it does not consider earlier measurements in its algorithm. Instead, the controller interpretation of JSQ can be viewed as a P-controller with infinite gain as it always redirects *all* requests to the shortest queue, thus maximizing its control input in order to reach its setpoint.

Viewing JSQ as a feedback queue length controller JSQ with setpoint defined in Equation (3.3), leads to a situation of controllers with conflicting setpoints, as the graceful degradation controllers in the servers have setpoints based on the response time goal. JSQ has no tuning parameters, but the simulation results suggest that it does change its behavior for different arrival rates λ, as the oscillations become much clearer in this case when $\lambda = 1500/s$. Note that the server controllers keep the same tuning throughout the simulation and that the results for the random strategy do not change with λ. One interpretation is that JSQ becomes a faster queue length controller with increasing λ, as it then gets more requests per second that can be routed to the shortest queue. In this simulation example, the JSQ feedback controller becomes approximately equally strong as the server controllers for $\lambda = 1500/s$, leading to large oscillations due to the conflicting setpoints. For $\lambda = 400/s$, the issue is almost completely mitigated due to the separation of time scales, i.e., the server controllers are significantly faster than JSQ here. These changes in behavior for JSQ with the incoming arrival rate make this issue of possible negative interactions with server controllers even more severe, as it might be difficult to find if it is only tested in a certain range where a separation of time scales occurs.

The setup of Figure 3.4 was utilized in a previous effort [Dürango et al., 2014] of designing load balancing strategies for servers using Brownout control strategies from [Klein et al., 2014]. Their approach was to feed metrics from the Brownout controllers to the load balancer, in order to set server weights in a weighted random-based strategy. While this Brownout-aware approach performed reasonably well, it is evident that the randomness still involved in the decisions hurt the predictability of the application. Interestingly enough, they found that the best performing non-Brownout-aware load balancing strategy was in fact JSQ. A possible reason for the opposite conclusions between the investigation in this thesis and [Dürango et al., 2014] is the involved server controllers. In Paper III, the controllers at the server level utilizes the cascaded Brownout structure from Paper I, whereas the evaluation in [Dürango et al., 2014] uses the considerably slower original Brownout controller [Klein et al., 2014]. Thus, in the latter case it is possible that the controller and JSQ had a large enough separation of time scales to avoid the oscillatory behavior observed in Paper III.

The proposed solution in this thesis to the controller interactions problem is to instead improve the architecture, such that the load balancing strategy and server controllers do not risk to make conflicting decisions, both due

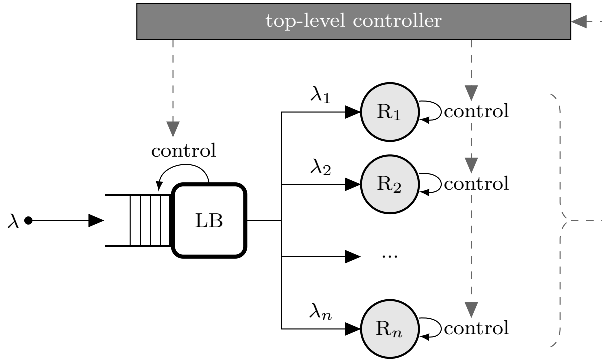


Figure 3.6 The proposed load balancing architecture, with the goal of mitigating negative interactions between decision-making layers. Figure from Paper III.

to conflicting setpoints and long delays between the decisions. The proposed architecture is shown in Figure 3.6, which exhibits a more hierarchical design compared to Figure 3.4. In this proposed setup, the queues have been removed from the servers and are instead replaced by a central queue at the load balancer, in order to minimize the delay between decisions at the two levels.

The load balancer now features a *waiting time* controller, that controls its queue length through a simple I-controller with the graceful degradation decision as its actuator. The server replicas now feature *service time* controllers that control the time the requests spend in the servers using an I-controller, actuated by deciding the number of concurrent requests. In order to control this concurrency number, each server controller piggy-backs the responses of completed requests with an order for new requests from the load balancer. In this way, the responsibilities of the load balancer and server controllers have been reversed compared to the setup in Figure 3.4. The routing decisions are now made by the server controllers themselves, which eliminates both possible negative interactions as well as the randomness involved in the Brownout-aware strategies proposed in [Dürango et al., 2014]. The structure is completed by a *top-level* controller, that ensures that the end-to-end response times, including both waiting and service time, follow the desired setpoint. Once again, this is performed by a simple I-controller, this time actuated by setting the setpoints of the lower level controllers.

The proposed structure enables design of almost trivial I-controllers, as their tasks at hand are simplified by the hierarchical structure. A possible drawback, however, is that the central queue at the load balancer might act as a bottleneck if the implemented logic does not execute fast enough. Also, the optimal concurrency numbers in the servers might differ from application to

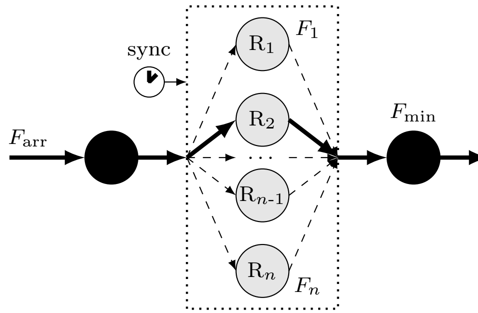


Figure 3.7 Synchronized service system. Figure from Paper IV.

application, which might limit the set of possible control inputs in the server controllers. It is thus possible, that the proposed structure might require some modifications in order to run in a real cloud setup. However, the simplicity of the involved controllers should enable a fast and efficient implementation at all levels.

3.3 Queuing-Theoretical Methods

The second method track is focused on applying queuing-theoretical concepts using request cloning as actuator. The cloud applications considered are similar to the ones in the control-theoretical method track, i.e., replicated over multiple servers. This track does not focus on obtaining dynamical models, instead the analysis is aimed towards stationary models that can be used for gaining insight into important predictability aspects of the application. In addition, the models are also useful for decision making.

Simplified Modeling for Request Cloning

Paper IV describes modeling concepts that enable a simplified way of analyzing request cloning for replicated cloud applications. The analysis is mainly performed for the processor sharing queuing discipline, in order to widen the scope of the previous work which mostly concerns the FCFS discipline. Additionally, the PS discipline is usually an accurate processing model of preemptible applications, and exhibits some interesting properties when used together with the JSQ load balancing strategy.

The first main contribution is the formalization of the enabling *synchronized service* criterion, that allows modeling of cloning using the minimum distribution theorem, described in Theorem 1 in Section 2.2. The concept is shown in Figure 3.7, where cloning is performed to all n servers in the system. In the figure, the request sent to server replica R_2 terminates first

and produces the response, while the other clones are cancelled. In order to guarantee synchronized service, the following two conditions need to be met:

1. All clones have to be sent simultaneously to all servers.
2. Perfect cancellation is required, i.e., the processing in all $n-1$ servers that did not produce the fastest response, needs to cancel immediately.

Note that synchronized service criterion does not imply *immediate* service. Clones of the same original request do not have to enter the processing in the servers immediately, and can queue at the servers. Synchronized service only requires that requests enter and leave the *processing* simultaneously, which means that the concept is compatible with any queuing discipline, given that it is the same across all servers.

Assuming that the synchronized service criterion is fulfilled, the fastest clone is also the one that received the shortest service time, as all other parameters are equal. This means that the service time distribution $F_{\text{min}}(x)$ for the request clones that deliver the response can be exactly determined using the minimum distribution theorem presented in Section 2.2. As a result, the server system with synchronized cloning shown in Figure 3.7 can be *equivalently* modeled as a single server, with the same unaffected inter-arrival distribution $F_{\text{arr}}(x)$ and the determined service time distribution $F_{\text{min}}(x)$, using the same queuing discipline as the n servers. As the minimum distribution theorem has no assumptions what so ever, this equivalent modeling can be performed for any (possibly heterogeneous and dependent) service time distribution under any queuing discipline, assuming that the synchronized service criterion is fulfilled.

A simple example is synchronized cloning to n heterogeneous and independent servers R_i under exponential service time distributions with service rates μ_i. Then, the equivalent model becomes a single server with service time distribution $F_{\text{min}}(x)$ calculated according to Equation (2.11):

$$F_{\text{min}}(x) = 1 - \prod_{i=1}^{n} \{1 - F_i(x)\} = 1 - \prod_{i=1}^{n} e^{-\mu_i x} = 1 - e^{-\sum_{i=1}^{n} \mu_i x}. \tag{3.4}$$

The equivalent model for this example is thus also exponential, with a service rate equal to the sum of all involved service rates.

The main advantage with the equivalent model is that it reduces the server system with cloning to a single server with a service time distribution that can be determined. Thus, all previous queuing-theoretical methods and results for a single server, where some are presented in Section 2.2, can be applied to analyze the server systems subject to cloning as well, i.e., there is no need for reinventing the wheel. Additionally, a system of n servers subject to cloning can also be divided into subsystems, or *clusters*, of m servers each,

where each subsystem guarantees synchronized service. Thus, the equivalent modeling procedure can be performed for each subsystem respectively, which reduces the cloned server system to a traditional server system of n/m servers. As a result, all previous methods and results for server systems can be applied as well.

The second main contribution is an analysis of imperfections, i.e., of situations where the synchronized service criterion does *not* hold. The analysis presented so far assumes perfect synchronized service, which is almost impossible to achieve in a real implementation. Thus, it is of high importance to investigate the effect of imperfections on the model as well. Two common imperfections that break the synchronized service criterion are analyzed in this thesis, (i) arrival delays, where clones reach the servers at different times due to, e.g., network issues; and (ii) cancellation delays, representing delays that can occur when the serving of requests are to be terminated.

The outcome of the analysis of the imperfections is in the form of error bounds, i.e., bounds on how the average response time of an imperfect system $\mathcal{S}_1$ compares to an otherwise identical system $\mathcal{S}_2$ fulfilling perfect synchronization. All proofs are available in Paper IV but are left out here for brevity. For arrival delays of average time $\mathbf{E}[a]$, the error bound for the model becomes

$$\mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2] + \mathbf{E}[a], \tag{3.5}$$

i.e., the errors in average response time are bounded by the average arrival delays. For average cancellation delays $\mathbf{E}[c]$ the error bound becomes

$$\mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2], \tag{3.6}$$

where $\mathcal{S}_2$ has service time $X|\mathcal{S}_2$ identical to

$$X|\mathcal{S}_2 = X|\mathcal{S}_1 + \mathbf{E}[c]. \tag{3.7}$$

In other words, the errors in the average response time due to cancellation errors $\mathbf{E}[c]$ are bounded by an otherwise identical, but synchronized system with $\mathbf{E}[c]$ as additional service time. Finally, the error bounds are also proven in Paper IV to be additive, i.e., the errors of a system $\mathcal{S}_1$ subject to both arrival and cancellation delays are bounded by the sum of the bounds presented in Equations (3.5)-(3.7). Utilizing these error bounds, the analysis presented for synchronized service can thus be extended to include more realistic systems subject to implementation imperfections.

Cloning and Speculative Execution for JSQ

Paper IV utilizes properties of the JSQ load balancing strategy to extend the synchronized clone-to-clusters methods. The concept of dividing the n servers into clusters, or subsystems, to achieve synchronization is slightly

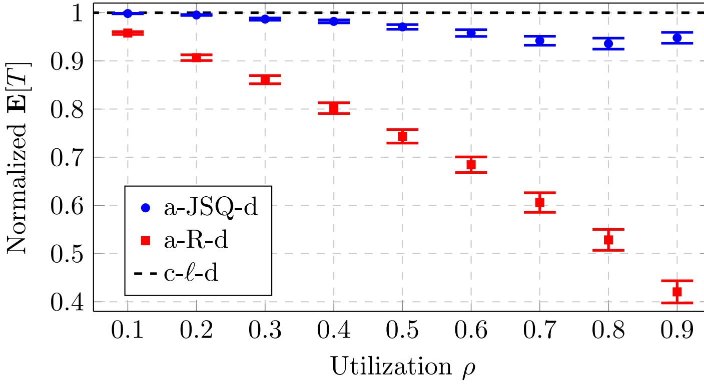


Figure 3.8 Simulations comparing a-ℓ-d to c-ℓ-d for random and JSQ. The intervals represent 95% confidence intervals. Figure from Paper IV.

cumbersome. The first main contribution in this area is thus to (i) introduce a clone-to-any concept, where clones can be sent to any server; and (ii) show that for the JSQ load balancing strategy under the PS discipline, this cloning strategy can be approximated well by the synchronized clone-to-clusters model.

The clone-to-any method breaks the synchronized service criterion by sending the clones of an original request to any m of the n servers in total, and these servers are then not guaranteed to be of the same queue length. However, as seen in the analysis on imperfections, the modeling accuracy is still high as long as the synchronization errors are small. The JSQ load-balancer was in Paper III interpreted as a queue length controller, successfully keeping the queue lengths of all n servers close to each other. For the processor sharing discipline investigated in Paper IV, there is no queuing involved for clones, rather they experience processor shares based on how many other requests they share their servers with during their processing time. As the JSQ load balancer acts to keep all queue lengths, or server occupancies, equal, then all clones of an original request will experience approximately equal processor shares. Thus, from a cloning perspective under the PS discipline, the JSQ load balancing strategy acts as a *service synchronizer*, providing all clones *near-synchronized* service. As a result, the clone-to-any method for JSQ with d number of clones, denoted as a-JSQ-d, can be expected to perform similarly to its synchronized clone-to-clusters counterpart, denoted as c-JSQ-d. As the latter can be modeled using the concept from Paper IV, the a-JSQ-d method can also be approximately modeled using the same methods.

Figure 3.8 shows results from simulations comparing two clone-to-any strategies a-ℓ-d to their clone-to-clusters counterparts c-ℓ-d. The simulations

were performed for different number of servers n and varying cloning factors d, and from these results 95% confidence intervals were formed. The normalization of $\mathbf{E}[T]$ was performed such that each value is divided by the value for the c-ℓ-d counterpart, thus a value close to 1 represents that they are approximately equal. The Random load balancing strategy a-R-d, included for reference, is obviously not well modeled by its c-R-d counterpart, unless for very low utilization. The JSQ results, on the other hand, stay within an error of 10% for any utilization, implying that a-JSQ-d can be approximately modeled by c-JSQ-d with high accuracy for all the evaluated cases. The model accuracy is, however, highest for low utilization, since many servers are empty during low loads. As JSQ is excellent in finding empty servers, almost all clones will run alone with the same processor share equal to 1, thus achieving near-perfect synchronization.

The second main contribution utilizing the JSQ discipline is presented in Paper V. Here, the near-synchronization properties of JSQ for cloning under the processor sharing discipline are used to extend the results obtained in Paper IV. As the JSQ discipline is efficient in keeping all server occupancies equal, the clones of the same original request do not even have to be sent out simultaneously, as they will still receive approximately equal processor shares. Thus, the same concept can be applied to approximately model speculative execution, where clones are sent after a known speculation time. Cloning can be interpreted as a special case of speculative execution with zero speculation time for all clones. Thus, extending this modeling concept to also include non-zero speculation time gives opportunities to find new configurations that outperform traditional cloning. However, as the search space of possible configurations increases from bounded to unbounded, it becomes more difficult to find the optimal combination of speculation times and cloning factors.

3.4 Common Themes

Apart from the common goal of increasing predictability of cloud applications, both method tracks share common concepts and ideas.

Design Choices for Simplification

The first concept that runs through the entire thesis is the goal of finding design choices that enable a simplified world view, without limiting the applicability of the proposed methods and strategies. Examples from the control-theoretical method track described in Section 3.2 include: (i) the feedback linearization-like modeling approach in Paper I for queue length dynamics; (ii) the simple, yet applicable model for queue dynamics under more general assumptions in Paper II; and (iii) the new proposed architecture for easier co-design of load balancing strategy and server controllers in Paper III. For

the queuing-theoretical track in Section 3.3 the main examples are: (i) the formalization of the synchronized service criterion in Paper IV that enables simplified modeling of request cloning under general assumptions; and (ii) the interpretation of JSQ as a near-synchronizer that allows for approximate, yet accurate modeling of an otherwise complex scenario, for both basic cloning in Paper IV and its extension speculative execution in Paper V.

Application of Traditional Concepts in New Domains

The second concept that can be found throughout the thesis is the method of applying well-known approaches from other field into new domains. This way of thinking sometimes requires to look at the problem from a slightly different angle than the state-of-the art of the current domain. The goal is still, however, that for the identified possibilities of applying methods from other domains, the solution once completed should feel as an integrated part of the new domain as well. Examples from both method tracks include: (i) the cascaded controller structure, usually found in chemical process control, here applied for response time control in Paper I; (ii) the interpretation of JSQ as a queue length controller in Paper III; and (iii) the utilization of the minimum distribution theorem, a more than a century old result from statistics, for the modeling of request cloning in Paper IV.

4

Future Work

This section discusses possibilities for improvements and extensions to the methods and strategies proposed in this thesis. Cloud computing is a changing and growing environment, where new concepts and trends emerge every year. The work that lays the foundation of this thesis started already in 2016, in a time where popular cloud concepts of today were only beginning to gain popularity. As a result, the research performed in this thesis could in the future be extended towards many newer areas within cloud computing, such as serverless and edge computing, as well as the microservices architecture. However, due to the high abstraction level approach of both modeling and decision-making in thesis, there should be great possibilities to adopt the findings presented to fit the purposes of future cloud technologies.

Regarding extensions and future research directly related to the work presented in this thesis, some examples deserve to be mentioned. First, from a control-theoretical perspective it would be interesting to further investigate how to explicitly model the JSQ load balancing strategy as a controller. With a more formalized approach, the patterns noticed in the interactions with the server controllers could be explained in greater detail, such as the frequency of the oscillations observed. Also, it could be possible to predict in what scenarios JSQ would interact more intensely with the server controllers and when the performance would be reasonable. Second, from a more queuing-theoretical perspective it would be interesting to look into additional possible applications where the proposed request cloning model could be used as a basis for approximate modeling. In other words, if it would be possible to extend beyond the investigated scenarios including delay imperfections and the near-synchronization property of JSQ. Finally, it would also be interesting to extend the evaluations performed in the simulated cloud environment of this thesis, towards systematical investigations implemented in a real cloud setup. As the models and decision-making strategies proposed in this thesis strive to rely on non-restricting assumptions, there should be potential for successful usage in real-world implementations as well. However, this of course remains to be empirically proven in the future.

Bibliography

Abdelzaher, T. F., K. G. Shin, and N. Bhatti (2002). “Performance guarantees for web server end-systems: A control-theoretical approach”. *IEEE Transactions on Parallel and Distributed Systems* **13**:1. ISSN: 1045-9219.

Alvarez-Ramirez, J., J. Alvarez, and A. Morales (2002). “An adaptive cascade control for a class of chemical reactors”. *International Journal of Adaptive Control and Signal Processing* **16**:10, pp. 681–701.

Ananthanarayanan, G., A. Ghodsi, S. Shenker, and I. Stoica (2013). “Effective straggler mitigation: Attack of the clones”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation.* nsdi’13. USENIX Association, Lombard, IL, pp. 185–198.

Arcelli, D., V. Cortellessa, A. Filieri, and A. Leva (2015). “Control theory for model-based performance-driven software adaptation”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA ’15.* ACM Press.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. (2010). “A view of cloud computing”. *Communications of the ACM* **53**:4, pp. 50–58.

Åström, K. J. and B. Wittenmark (1997). *Computer-controlled Systems (3rd Ed.)* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN: 0-13-314899-8.

Åström, K. J. and T. Hägglund (2006). *Advanced PID control.* Vol. 461. ISA-The Instrumentation, Systems, and Automation Society Research Triangle Park.

Åström, K. J. and R. M. Murray (2019). *Feedback Systems: An Introduction for Scientists and Engineers, Second Edition.* Princeton University Press, Princeton, NJ.

Balalaie, A., A. Heydarnoori, and P. Jamshidi (2015). “Migrating to cloud-native architectures using microservices: An experience report”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer, pp. 201–215.

Barroso, L. A. and U. Hölzle (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool.

Bayramusta, M. and V. A. Nasir (2016). “A fad or future of IT?: A comprehensive literature review on the cloud computing research”. *International Journal of Information Management* **36**:4, pp. 635–644.

Bokhari, M. U., Q. M. Shallal, and Y. K. Tamandani (2016). “Cloud computing service models: A comparative study”. In: *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 890–895.

Bolton, W. (2021). *Instrumentation and control systems*. Newnes.

Braake, H. A. te, E. J. Van Can, J. M. Scherpen, and H. B. Verbruggen (1998). “Control of nonlinear chemical processes using neural models and feedback linearization”. *Computers & chemical engineering* **22**:7-8, pp. 1113–1127.

Brodkin, J. (2013). “Facebook opens data center filled entirely with servers it designed”. URL: `https://arstechnica.com/information-technology/2013/06/facebook-opens-data-center-filled-entirely-with-servers-it-designed/`.

Burns, B., J. Beda, and K. Hightower (2018). *Kubernetes*. Dpunkt Heidelberg, Germany.

Cerf, S., M. Berekmeri, B. Robu, N. Marchand, and S. Bouchenak (2016). “Cost function based event triggered model predictive controllers application to big data cloud services”. In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, pp. 1657–1662.

Chandy, K. M., U. Herzog, and L. Woo (1975). “Approximate analysis of general queuing networks”. *IBM Journal of Research and Development* **19**:1, pp. 43–49.

Chiasson, J. (1998). “A new approach to dynamic feedback linearization control of an induction motor”. *IEEE Transactions on Automatic Control* **43**:3, pp. 391–397.

Craig, I., C. Aldrich, R. Braatz, F. Cuzzola, E. Domlan, S. Engell, J. Hahn, V. Havlena, A. Horch, B. Huang, et al. (2011). “Control in the process industries”. *The impact of control technology. IEEE control systems society*.

Dean, J. and L. A. Barroso (2013). “The tail at scale”. *Communications of the ACM* **56**:2, p. 74.

Ding, S., S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas (2011). "Indexing strategies for graceful degradation of search quality". In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pp. 575–584.

Dürango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with Brownout". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. CDC14, pp. 5320–5327.

Gan, Y. and C. Delimitrou (2018). "The architectural implications of cloud microservices". *IEEE Computer Architecture Letters* **17**:2, pp. 155–158.

Gan, Y., Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. (2019). "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 3–18.

Gardner, K., S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia (2015). "Reducing latency via redundant requests: Exact analysis". *ACM SIGMETRICS Performance Evaluation Review* **43**:1, pp. 347–360.

Ghomi, E. J., A. M. Rahmani, and N. N. Qader (2017). "Load-balancing algorithms in cloud computing: A survey". *Journal of Network and Computer Applications* **88**, pp. 50–71.

Guo, H., Y. Liu, G. Liu, and H. Li (2008). "Cascade control of a hydraulically driven 6-dof parallel robot manipulator based on a sliding mode". *Control Engineering Practice* **16**:9, pp. 1055–1068.

Gupta, V., M. H. Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9-12, pp. 1062–1081.

Gupta, V., M. Harchol-Balter, J. Dai, and B. Zwart (2010). "On the inapproximability of M/G/K: Why two moments of job size distribution are not enough". *Queueing Systems* **64**:1, pp. 5–48.

Hall, S. (2017). *Rules of thumb for chemical engineers*. Butterworth-Heinemann.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2004). *Feedback Control of Computing Systems*. John Wiley & Sons. ISBN: 047126637X.

Jain, N. and S. Choudhary (2016). "Overview of virtualization in cloud computing". In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, pp. 1–4.

Jalaparti, V., P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan (2013). “Speeding up distributed request-response workflows”. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM ’13. ACM, Hong Kong, China. ISBN: 978-1-4503-2056-6.

Jean-Marie, A. and P. Robert (1994). “On the transient behavior of the processor sharing queue”. *Queueing Systems* **17**:1-2, pp. 129–136.

Jonas, E., J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. (2019). “Cloud programming simplified: A Berkeley view on serverless computing”. *arXiv preprint arXiv:1902.03383*.

Joshi, G., E. Soljanin, and G. Wornell (2015). “Efficient replication of queued tasks for latency reduction in cloud systems”. In: *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE.

Kendall, D. G. (1953). “Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain”. *The Annals of Mathematical Statistics*, pp. 338–354.

Kjaer, M. A., M. Kihl, and A. Robertsson (2009). “Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers”. *IEEE Transactions on Network and Service Management* **6**:4. ISSN: 1932-4537.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). “Brownout: Building more robust cloud applications”. In: *36th International Conference on Software Engineering*. ICSE14. ACM, Hyderabad, India, pp. 700–711. ISBN: 978-1-4503-2756-5.

Kleinrock, L. (1975). *Queueing Systems*. Vol. I: Theory. Wiley Interscience.

Konstanteli, K., T. Cucinotta, K. Psychas, and T. Varvarigou (2012). “Admission control for elastic cloud services”. In: *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 41–48.

Leva, A. (2018). “PID-based controls in computing systems: A brief survey and some research directions”. *IFAC-PapersOnLine* **51**:4, pp. 805–810.

Little, J. D. (1961). “A proof for the queuing formula: $L = \lambda W$”. *Operations research* **9**:3, pp. 383–387.

Ljung, L. (2000). *Control theory: Multivariable and nonlinear methods*. Taylor & Francis.

Lorido-Botran, T., J. Miguel-Alonso, and J. A. Lozano (2014). “A review of auto-scaling techniques for elastic applications in cloud environments”. *Journal of grid computing* **12**:4, pp. 559–592.

Maggio, M., H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva (2012). "Comparison of decision-making strategies for self-optimization in autonomic computing systems". *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **7**:4, pp. 1–32.

Maggio, M., C. Klein, and K.-E. Årzén (2014). "Control strategies for predictable brownouts in cloud computing". *IFAC Proceedings Volumes* **47**:3. 19th IFAC World Congress. ISSN: 1474-6670.

Mao, M. and M. Humphrey (2012). "A performance study on the VM startup time in the cloud". In: *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 423–430.

Masanet, E., A. Shehabi, N. Lei, S. Smith, and J. Koomey (2020). "Recalibrating global data center energy-use estimates". *Science* **367**:6481, pp. 984–986.

Medel, V., O. Rana, J. Á. Bañares, and U. Arronategui (2016). "Modelling performance & resource management in Kubernetes". In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pp. 257–262.

Mell, P. and T. Grance (2011). "The NIST definition of cloud computing".

Millnert, V. and J. Eker (2020). "HoloScale: Horizontal and vertical scaling of cloud resources". In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 196–205.

Millnert, V., J. Eker, and E. Bini (2017). "Dynamic control of NFV forwarding graphs with end-to-end deadline constraints". In: *2017 IEEE International Conference on Communications (ICC)*. IEEE, pp. 1–7.

Newman, S. (2021). *Building microservices*. O'Reilly Media, Inc.

Nylander, T., C. Klein, K.-E. Årzén, and M. Maggio (2018). "Brownout^{CC}: Cascaded control for bounding the response times of cloud applications". In: *2018 American Control Conference*. Milwaukee, Wisconsin, USA.

Paganini, F., A. Tang, A. Ferragut, and L. L. H. Andrew (2012). "Network stability under alpha fair bandwidth allocation with general file size distribution". *IEEE Transactions on Automatic Control* **57**:3, pp. 579–591.

Qiu, Z., J. F. Pérez, and P. G. Harrison (2016). "Tackling latency via replication in distributed systems". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*. ACM Press.

Rugh, W. J. and J. S. Shamma (2000). "Research on gain scheduling". *Automatica* **36**:10, pp. 1401–1425.

Saleem, A., B. Taha, T. Tutunji, and A. Al-Qaisia (2015). “Identification and cascade control of servo-pneumatic system using particle swarm optimization”. *Simulation Modelling Practice and Theory* **52**, pp. 164–179.

Sharma, S., S. Singh, and M. Sharma (2008). “Performance analysis of load balancing algorithms”. *World academy of science, engineering and technology* **38**:3, pp. 269–272.

Shehabi, A., S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, and W. Lintner (2016). “United States data center energy usage report”.

Smith, J. E. and R. Nair (2005). “The architecture of virtual machines”. *Computer* **38**:5, pp. 32–38.

Soltesz, S., H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson (2007). “Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors”. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys european conference on computer systems 2007*, pp. 275–287.

U.S. Department Of Energy (2020). “Annual energy outlook 2020”.

Ueda, T., T. Nakaike, and M. Ohara (2016). “Workload characterization for microservices”. In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–10.

Varghese, B. and R. Buyya (2018). “Next generation cloud computing: New trends and research directions”. *Future Generation Computer Systems* **79**, pp. 849–861.

Wang, W.-P., D. Tipper, and S. Banerjee (1996). “A simple approximation for modeling nonstationary queues”. In: *Proceedings of IEEE INFOCOM ’96. Conference on Computer Communications*. IEEE Comput. Soc. Press.

Widder, D. V. (2015). *Laplace transform (PMS-6)*. Princeton university press.

Wittenmark, B., K. J. Åström, and K.-E. Årzén (2002). “Computer control: an overview”. *IFAC Professional Brief* **1**, p. 2.

Wolff, E. A. and S. Skogestad (1996). “Temperature cascade control of distillation columns”. *Industrial & engineering chemistry research* **35**:2, pp. 475–484.

Xing, Y. and Y. Zhan (2012). “Virtualization and cloud computing”. In: *Future Wireless Networks and Information Systems*. Springer, pp. 305–312.

Zhang, J., J. Dai, and B. Zwart (2009). “Law of large number limits of limited processor-sharing queues”. *Mathematics of Operations Research* **34**:4, pp. 937–970.

Paper I

Brownout^{CC}: Cascaded Control for Bounding the Response Times of Cloud Applications

Tommi Nylander **Cristian Klein** **Karl-Erik Årzén**
Martina Maggio

Abstract

Cloud computing has emerged as an inexpensive and powerful computing paradigm, to the point that now even applications with hard deadlines are executed in the cloud. It may happen, due to unexpected events, that an application becomes popular and receives a lot of attention and client requests in a short period of time. Provisioning computing capacity for such applications is quite a difficult task, because content popularity cannot be easily predicted. One of the main problems in case content has to be served with a hard deadline is to ensure that this deadline is respected, even in the presence of popularity spikes. To this end, partial computation and graceful degradation were exploited, originating the *brownout* framework. Applications would degrade the user experience in the presence of load variations, to guarantee that deadlines are met. Two different control paradigms were applied to brownout: discrete-time control of optional content percentage over a period and event-based queue management. The first one had reasonable performance providing formal guarantees about the solution. The second one was able to improve the performance and keep the response time at the setpoint better, but suffered from the drawback of not providing formally-grounded mathematical guarantees. In this work we combine the best of both worlds, providing a cascaded controller for brownout, based on a more precise model of the cloud application with respect to the original design. The Brownout^{CC} controller achieves performance comparable with the event-based version, without sacrificing formal guarantees.

© 2018 AACC. Originally published in American Control Conference (ACC), Milwaukee, July 2018. Reprinted with permission. The article has been reformatted to fit the current layout.

1. Introduction

Control theory is becoming important in domains where problems were previously solved using heuristic solutions, without having access to formally grounded analysis tools. One of these is the computing systems domain [Hellerstein et al., 2004]. Computing resource allocation can easily be cast into a control problem, where a controller decides the amount of resource to allocate to different entities based on desired and measurable performance metrics [Yun and Proutiere, 2015; Kjaer et al., 2009; Lu et al., 2002; Abdelzaher et al., 2002]. Recently, the cloud computing domain has emerged as an interesting application domain for control-theoretical principles and techniques [Maggio et al., 2014; Durango et al., 2014; Barna et al., 2016; Deliparaschos et al., 2016; Kalyvianaki et al., 2014].

One of the most difficult problems in cloud computing is to quickly and effectively react in the presence of flash crowds. A flash crowd is caused by a sudden increase in popularity of some content, that is then served to millions of users at the same time. The amount of resources needed to serve this increased amount of requests is unlikely to be available, unless there was a substantial over-provisioning of computing capacity before the raise in popularity.

To mitigate this problem, it is common to resort to techniques like graceful degradation. A possible way of degrading the performance of a web server is to deny admission to some of the requests when it is not possible to meet the user demands [Abdelzaher et al., 2002]. Admission control means that some users would not receive any response at all, hence risking losing them to competitors, incurring long-term revenue loss. Another possibility is to assign a maximum time to each request and iteratively refine an answer until the time budget expires [Ding et al., 2011; Jalaparti et al., 2013]. This strategy works well for pruning search queries of spurious results, but does not easily generalize to all types of cloud applications.

A third possibility to apply the principles of graceful degradation is called *brownout* [Maggio et al., 2014; Klein et al., 2014]. When producing the response to the user requests, it is often possible to identify a part of the response that is the necessary information the user wants to see and a part of the response that would provide a better user experience and increased revenues, but is not mandatory. In the case of a travel agency website, the mandatory part of the response is the flight search, while additional optional information are car rental locations and hotel suggestions. Clearly, the application owner wants to provide the additional information, but not at the expense of losing a customer. Brownout [Klein et al., 2014] divides the response into the two mentioned parts and measures the response time to determine if the optional content is served (at an additional computation cost) or not.

The core idea behind brownout is to serve as much optional content as possible, without penalizing response times. The cloud application uses feedback from the response times to determine how much optional content can be served without sacrificing performance. The first brownout proposal used a very simple first-order model for the system [Klein et al., 2014; Maggio et al., 2014] and proposed some control strategies. Using discrete-time control, it was possible to prove properties of the closed-loop system, like stability and zero steady-state error [Klein et al., 2014]. However, the sampling period of the controller would still be a critical parameter. Decisions would be made periodically, but depending on the arrival rate of requests at the server, the control period could either be too small, leading to taking decisions based on too few response times, or too large, leading to a large lag in controller response. This could mean deciding based on the average of many response times or of none. To avoid this disparity and gain additional performance, an event-based version of the brownout principle was then devised [Desmeurs et al., 2015], which would take a new decision at every request arrival. The event-based brownout controller [Desmeurs et al., 2015] showed very good performance, but lacked the mathematical formalization and analysis possibility.

The contribution of this paper is three-fold: (i) We formalize the brownout control strategy in [Desmeurs et al., 2015] into a cascaded control problem; (ii) We design the inner and outer loop controllers, proposing both a feedback and a feedforward plus feedback version. Our controller features both the performance of [Desmeurs et al., 2015] and the formal guarantees of [Klein et al., 2014]; (iii) We evaluate our approach and compare with previous solutions using the *brownout* simulator. Besides providing formal guarantees, our controllers show fewer oscillations and maintain the measured response times closer to the target.

2. The brownout approach

This section provides some background information about the brownout model and controllers developed in [Klein et al., 2014; Desmeurs et al., 2015]. It also introduces some basic terminology that will then be used to explain the Brownout$^{\text{CC}}$ approach.

A brownout-aware application generates responses that are composed of two different parts: the mandatory and the optional content. In some cases, a response is produced including both parts of the content, while in other cases, to speed up the process and consume less resources, only the mandatory part is included in the response. The aim of the brownout approach is to maintain certain statistics for the user response time. In cloud computing, the focus is on maintaining tail response time – instead of average – as it was shown

to better correlate with user experience [Dean and Barroso, 2013]. For this reason, we focus our effort on the 95$^{\text{th}}$ percentile of the response time for the user requests.

Furthermore, notice that simplicity is an important feature of every control strategy for a system like this. In fact, the control computation happens on the same hardware that provides responses to the users' requests. In case the control strategy is simple and executes fast enough, more hardware power is devoted to answering requests from actual users of the web application. Due to this remark, simplicity is one of the key points in evaluating our control strategy.

This simplicity also applies to plant modeling. In contrast to physical plants, the hardware and software stack of cloud applications are so complex that it is unfeasible to devise a detailed model. Therefore, when controlling software system, one aims for as simple plant models as possible while still capturing the essential relationship between inputs and outputs. This also implies that linear models and linear design techniques often are a good choice.

2.1 Original control strategy

Assume that a brownout controller is periodically selecting the probability of including the optional content in a response, called the *dimmer* value. The controller period is τ_c seconds and to each controller intervention we associate a cardinal number k. We denote by $\theta(k)$ the dimmer value that the controller computes for the interval $[(k-1)\,\tau_c, k\,\tau_c]$.

The brownout approach presented in [Klein et al., 2014] assumes that the cloud application behaves according to a very simple first-order model. According to the model, the value of the 95$^{\text{th}}$ percentile of the response time τ_{95} varies depending on the dimmer value as follows

$$\tau_{95}(k) = \phi(k-1)\,\theta(k-1) + \delta\tau_{95}(k), \tag{1}$$

where $\phi(k-1)$ is a time-varying coefficient that depends on the computing platform and can be estimated and $\delta\tau_{95}(k)$ is a disturbance, interfering with the nominal system's behavior. Loop shaping is then used to synthesize a controller for the system. We denote by $e_{\tau 95}(k)$ the error between the desired 95$^{\text{th}}$ percentile of the response time $\bar{\tau}_{95}(k)$ and the actual value. Assuming that no disturbance is acting on the system, the desired closed loop system Z-transform between the setpoint $\bar{\tau}_{95}(k)$ and the actual value of $\tau_{95}(k)$ is

$$G(z) = \frac{1-p_b}{z-p_b} \tag{2}$$

where p_b, the pole of the closed loop system, is simply a parameter of the

controller. The unsaturated dimmer value $\theta^*(k)$ can then be selected as

$$\theta^*(k) = \theta(k-1) + \frac{1 - p_b}{\hat{\phi}(k)} e_{\tau 95}(k) \tag{3}$$

where $\hat{\phi}(k)$ is an estimate of $\phi(k)$ obtained with a Recursive Least Square (RLS) filter. The dimmer value θ represents the probability of carrying out the execution of the optional content, therefore it is saturated in order to be bounded in the interval $[0, 1]$.

The expression of the closed loop system in (2) allows one to prove stability (provided that the pole p_b is chosen accordingly) and zero steady-state error (the static gain is equal to 1). The proof is subject to how well the model (1) approximates the behavior of the cloud application [Klein et al., 2014].

2.2 Event-driven brownout

The event-based version of the brownout paradigm [Desmeurs et al., 2015] works as follows. A periodic controller updates a threshold value $\psi_q(k)$ for the length of the queue of requests that have not yet been answered, with period τ_c.

Assume that a request r arrives at time t_r and that time t_r is included in the control interval $[k, k+1]$. Denote with $o_r \in \{0, 1\}$ the indicator of the execution of the code for the optional content – i.e., if $o_r = 1$ the optional content is computed and if $o_r = 0$ the optional content is not computed. The web server compares the amount of requests already queuing in the system $q(t)$ and the threshold set by the controller at the closest k-th control period $\psi_q(k)$ and determines if the optional content should be provided or not.

$$\begin{array}{lcl} q(t_r) \geq \psi_q(k) & \Longrightarrow & o_r = 0 \\ q(t_r) < \psi_q(k) & \Longrightarrow & o_r = 1 \end{array} \tag{4}$$

This algorithm has the advantage of being very easy to implement. The threshold ψ_q was in [Desmeurs et al., 2015] set using a manually tuned PI controller with anti-windup.

However, the absence of a proper model for the queue behavior and the application behavior creates difficulties in proving properties of the closed loop system. Empirically though, the cloud application was shown to have very good performance in terms of the 95$^{\text{th}}$ percentile of the response times being close to its desired value [Desmeurs et al., 2015].

3. The Brownout^{CC}approach

This section describes the design of a brownout control strategy that combines the advantages of both the methods described in Section 2, obtaining

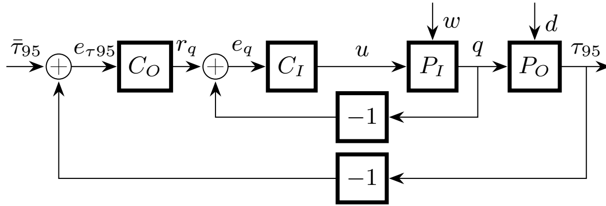


Figure 1. A general cascaded control structure interpretation of [Desmeurs et al., 2015].

a formally analyzable controller. Subsection 3.1 motivates the use of a cascaded structure. Section 3.2 describes the inner loop, while Sections 3.3, 3.4, and 3.5 respectively discuss modeling, feedback, and feedforward control of the outer loop.

3.1 Event-driven brownout interpreted as cascaded control

In this section, we take a closer look at the event-driven approach in [Desmeurs et al., 2015], that we briefly summarized in Section 2.2. We here show that the threshold-based algorithm described in Equation (4) – that decides on optional content execution via the variable o_r – can be interpreted as part of a queue length control loop. In this interpretation, the threshold $\psi_q(k)$ translates to a queue length setpoint $r_q(k)$. In fact, the threshold-based approach serves optional content until the threshold $\psi_q(k)$ is reached and avoids serving optional content when the threshold is passed. The number of enqueued requests is then kept as close as possible to a function of the threshold, therefore translating it into a setpoint $r_q(k)$.

We denote by t_r the arrival time of a generic request r. At t_r, the algorithm shown in Equation (4) tries to keep the measured queue length $q(t_r)$ equal to a setpoint $r_q(t_r)$, by means of a simple on/off controller – i.e., turning on and off the computation of the optional part of the response. The controller takes as input the queue length error $e_q(t_r) = r_q(t_r) - q(t_r)$, and determines the choice of executing optional content $o_r \in \{0, 1\}$ as control signal.

This queue length control loop is driven by the request arrival, and acts at times t_r, when the request is received. To fully describe the algorithm of Section 2.2, we need to complement this choice with the selection of the setpoint r_q, which as stated before, was done using a periodically executed PI controller.

The overall scheme can then be described using the cascaded structure depicted in Figure 1. In this representation, the generic control signal u (Figure 1), is the control signal o_r, C_I corresponds to the on/off controller in Equation (4) and C_O the manually tuned PI controller that selects the queue length setpoint. In [Desmeurs et al., 2015], P_I and P_O are left unmodeled.

The cascaded interpretation in Figure 1 lays the foundation for our approach. The generic inner loop control signal u influences the response times of the cloud application, by changing the length of the queue of unserviced requests. P_I is the transfer function from the control signal determined by the controller C_I to the queue length, while P_O models the effect of the queue length on the response times. The outer loop control signal $r_q > 0$ is determined by the outer controller C_O and indicates a queue length setpoint.

To complete the model, we introduce two terms – w and d – representing disturbances acting respectively on the inner and outer loop. A web application hosted in the cloud is always subject to disturbances, such as changes in the number of users or in the computation speed. For example, additional load could be co-located with the virtual machine hosting the application, changing the efficiency of the computation resources [Mars et al., 2011]. We distinguish between two different types of disturbances: w represents a disturbance that causes the queue length q to vary due to stochastic variations (i.e. deviations from the mean) in the arrivals, d, on the contrary, is a load disturbance that causes τ_{95} to deviate even if r_q is kept constant. The control strategy, i.e., C_I and C_O, should be designed with both disturbance types in mind, in order to successfully keep τ_{95} close to its setpoint $\bar{\tau}_{95}$.

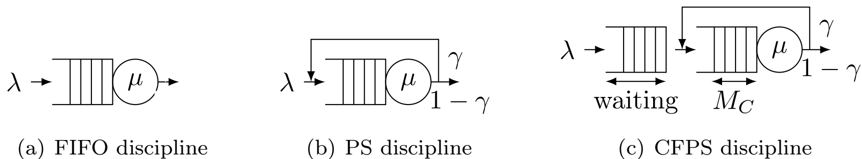
Viewing the control structure as a cascaded one has several advantages compared to single loop structure: (a) The system is faster in rejecting disturbances w acting on the inner loop; (b) The dynamics of the inner closed-loop can be linearized as shown in Section 3.2; and (c) The separation of the time-scales simplifies the control design. The inner controller can be designed to reject w disturbances of a fast stochastic nature, and the outer controller can be designed to reject load disturbances d. As a drawback, the cascaded structure requires measurements of the queue lengths in addition to the response time data. However, this is easy to solve from an implementation standpoint, as all the needed variables are already used in the implementation provided with [Desmeurs et al., 2015].

Motivated by the good performance obtained empirically with the event-based brownout controller despite the lack of modeling, and by the promising benefits of the structure, the Brownout^{CC} approach uses a model-based cascaded controller design and splits the modeling of the cloud application behavior into the two introduced loops.

3.2 Inner loop modeling and control

In cloud computing, usually applications are modeled using principles from queuing theory [Kleinrock, 1975]. We summarize in the following the background notions that inspired us in the design of the model and controller for the inner loop.

Queuing discipline models such as first-in-first-out (FIFO) and processor



(a) FIFO discipline (b) PS discipline (c) CFPS discipline

Figure 2. Queuing discipline models.

sharing (PS) are commonly used to model the behavior of web servers, see for example [Hoflack et al., 2008; Hellerstein et al., 2004; Cao et al., 2003; Gupta et al., 2007]. With the FIFO model, each request is executed individually based on the order of arrival, as represented by Figure 2(a). In the PS model, all the active requests are assumed to be executed simultaneously, using fractions of the computing capacity of the web server. The PS discipline can be seen as a queue where each request is processed for an (infinitely) short time-slice, and returned to the back of the queue, unless completed. From the modeling perspective, a queue that uses the PS discipline is normally seen as a queue with feedback where the single parameter, γ, represents the proportion of requests returned to the queue, as shown in Figure 2(b). A third option is the use of an approach that integrates both disciplines, the Combined FIFO and Processor Sharing (CFPS) model [Kjaer et al., 2007]. Here, the PS queue can only hold a limited $M_C > 0$ jobs. M_C models the number of available computing entities in the computing infrastructure – number of cores, number of threads – that can be executed in parallel. Requests exceeding M_C wait in a FIFO queue. This situation is shown in Figure 2(c). The CFPS model is a generalization of both FIFO and PS. These two disciplines are easily interpreted as special cases of CFPS, respectively with $M_C = 1$ and $M_C = \infty$.

For our approach to be as general as possible, we consider our application to behave as a queue with the CFPS discipline as the underlying model, without any restrictions on the value of M_C. We also avoid considering special arrival processes $A(t)$ or service time distributions $B(x)$, i.e., a G/G/1 queue.

To design a *proper* control strategy for the cascaded controller, we need a valid model for P_I in the form of a transfer function, that represents the behavior of the application queue length as a response to the control signal – $u = \theta$ in the case of the original controller [Klein et al., 2014] and $u = o_r$ for the event-based version [Desmeurs et al., 2015]. Writing such a (linear) model using queuing principles is difficult.

Here we use queuing theory as an inspiration to select a meaningful continuous-time control signal u that would allow us to model the inner loop plant P_I using a transfer function. We define $u = v = \mathrm{d}q/\mathrm{dt}$, representing the growth rate of the queue. Using this control signal, the transfer function

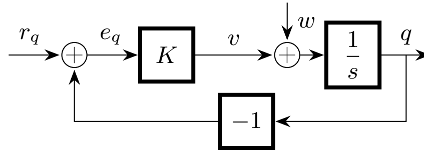


Figure 3. Inner loop control using feedback linearization.

$P_I(s)$ from v to q becomes a simple integrator:

$$P_I(s) = \frac{1}{s}. \tag{5}$$

By utilizing the concept of feedback linearization [Khalil, 1996], i.e., determining the choice of v and designing $C_I(s)$, we are able to linearize the inner loop and choose its dynamics. The dynamics of the closed inner loop $G_I(s)$ will affect the outer loop, leading to a desire for simplicity. To achieve this simple dynamics for $G_I(s)$, $C_I(s)$ is then chosen as a P controller with gain K:

$$C_I(s) = K. \tag{6}$$

As the process $P_I(s)$ is integrating, this simple controller is able to follow reference step changes in r_q without any stationary errors. However, these might still occur due to disturbances w entering the inner loop. The inner closed loop $G_I(s)$ becomes:

$$G_I(s) = \frac{K}{s+K}, \tag{7}$$

where the design parameter K determines the speed of the system. The complete inner loop model is shown in Figure 3.

We have now defined how to compute the control signal v. In order to complete the inner loop control, we should also specify how to *actuate* it. Our controller is realized using a periodic sampling strategy, with the actuation relying on the threshold-based algorithm (4). For each sampling period h:

(i) At the beginning of the sampling period h, i.e., at time t_a, the controller (6) calculates a control signal $v(t_a)$. The control signal represents the derivative of the queue length that we desire to actuate;

(ii) A queue length threshold $\psi_q(t_a)$ is set as: $\psi_q(t_a) = q(t_a) + v(t_a)$;

(iii) For all incoming requests during h, the algorithm in Equation (4) is used, for each request, to determine if optional content should be served or not;

(iv) This strategy ensures[1] that the new queue length $q(t_a + h)$ stays close to $q(t_a) + v(t_a)$, actuating $v(t_a)$.

On the negative side, the actuation strategy is not exact, i.e., it does not guarantee to exactly actuate v, as, e.g., the arrivals $A(t)$ enter the queue according to some general random process. These deviations from the intended queue growth rate caused by actuation errors can be seen as part of the disturbance w, entering as shown in Figure 3. On the positive side, the algorithm above actuates the control signal v well, regardless of both M_C, arrival process, and service time distribution. It also reacts quickly to stochastic changes in the system, like modifications of the arrival rate – thanks to its event-driven execution. Finally, it is also very simple to implement and requires minimal execution time.

After testing the inner controller in simulations, using different values of M_C, we choose $K = 1$ as the best fit for the inner loop design. The closed inner loop then becomes:

$$G_I(s) = \frac{K}{s+K} = \frac{1}{s+1}. \tag{8}$$

3.3 Outer loop modeling

To describe the outer open loop $G_P(s)$, i.e. from r_q to the response times, we split the model into two parts: (i) from r_q to q and; (ii) from q to the response times. The first part is completely described by the inner closed loop $G_I(s)$.

To model the second part we need to define precisely the meaning of "response times". We denote by τ_{95}^m the 95th percentile of the response times served only with mandatory content and by τ_{95}^o the 95th percentile of the response times of the requests served with mandatory and optional content. The mandatory τ_{95}^m and optional τ_{95}^o response times are expected to diverge depending on the value of M_C. The larger M_C becomes, the more the requests spend time being processed in the PS queue rather than waiting in the FIFO queue. As the mean service times are assumed to be related as $\bar{x}_m \ll \bar{x}_o$, a high value of M_C causes the mandatory τ_{95}^m and optional τ_{95}^o response times to diverge. Since we can only act on the optional response times, we measure and use for feedback only the optional response times τ_{95}^o.

[1] Assume that the mean inter-arrival times are denoted by $\bar{t}$, the mean mandatory and optional content service times respectively by $\bar{x}_m$ and $\bar{x}_o$, and that $\bar{x}_m < \bar{t} < \bar{x}_o$ holds. If the last assumption does not hold, brownout cannot find a feasible solution, and the inter-arrival times have to be adjusted to fit this assumption by e.g. adding or removing servers. According to Equation (4), mandatory content $o_r = 0$ is chosen for all $q(t_r) > \tau_q(k)$. Then, the queue length is "stable", i.e., kept close to (or slightly above) the threshold $\tau_q(k)$, since $\bar{x}_m < \bar{t}$. For a proof, see [Kleinrock, 1975]. Since optional content $o_r = 1$ is chosen for all $q(t_r) \leq \tau_q(k)$, the queue stays within a bound, ξ, around $\tau_q(k)$ since $\bar{x}_o > \bar{t}$ below the threshold.

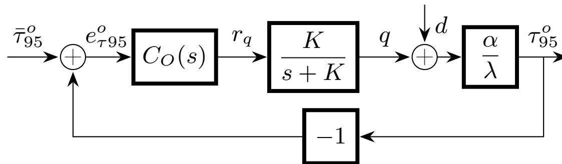


Figure 4. The outer loop model.

Then, the model of the second part, i.e., from q to τ_{95}^o, corresponding to the P_O block in Figure 1, can be inspired by Little's Law $\bar{\tau} = \bar{q}/\lambda$ [Kleinrock, 1975]. Here $\bar{\tau}$ and $\bar{q}$ represent mean response times and queue lengths, and λ represents the mean arrival rate. Instead of mean values, we want to model the 95$^{\text{th}}$ percentile of the response times. The theorem is thus not directly applicable, but it serves as a good approximation when we introduce a correction term, that we denote by α. The following static relation from q to τ_{95}^o is then proposed:

$$P_O(s) = \frac{\alpha}{\lambda}. \tag{9}$$

Here the constant α is assumed to vary with λ and M_C. The complete outer loop model is shown in Figure 4.

3.4 Design of outer loop feedback controller

The task is to design the outer loop controller $C_O(s)$, given the open loop transfer function from r_q to τ_{95}^o as

$$G_P(s) = \frac{K}{s+K}\frac{\alpha}{\lambda} = \frac{1}{s+1}\frac{\alpha}{\lambda}, \tag{10}$$

using $K = 1$ as chosen in Section 3.2.

We design the controller using pole-placement. As $G_P(s)$ is a first order system, the poles can be placed arbitrarily using only two controller parameters. In addition, the controller should be able to reject load disturbances d, resulting from stationary errors in the inner loop as well as from changes in the load. Furthermore, the controller should be able to handle the fact that α is unknown and varying and cope with changes in the process gain $G_P(0)$, especially since the arrival rate λ is expected to vary over time. The proposed solution is to select the controller parameters assuming a nominal process gain G_N. The adaptive controller gain k_a is then adjusted in order to counteract multiplicative changes to $G_P(0)$, such that $G_P(0)\,k_a \approx G_N$, giving the adaptive PI controller:

$$C_O(s) = k_a\left(k_p + \frac{k_i}{s}\right). \tag{11}$$

Here, $k_a = G_N/\hat{G}_P(0)$, where $\hat{G}_P(0)$ is estimated as described in Section 4. As $G_P(0)$ might change rapidly, it is not certain that k_a is able to adapt accordingly. Also, other model uncertainties might occur, requiring a robust design. For the nominal design, $\alpha = 1$ and $\lambda = 20$ are chosen giving $G_N = 0.05$, and the nominal process $G_P^N(s)$ as

$$G_P^N(s) = \frac{0.05}{s+1}. \tag{12}$$

The poles of the outer closed loop system are placed according to the characteristic equation

$$s^2 + 2\zeta\omega_O s + \omega_O^2, \tag{13}$$

where $0 \le \zeta \le 1$ is the relative damping and ω_O the speed of the outer loop. In order to ensure a robust design, $\zeta = 1$ is chosen placing the poles on the negative real axis as $(s + \omega_O)^2$. The choice of ω_O results in a trade off between robustness and noise rejection as the maximum M_S of the sensitivity function S in this case decreases when ω_O grows. As a result, $\omega_O = 0.6$ is chosen, setting the speed of the outer loop to about half the speed of the inner loop ($\omega_I = 1$). The choice also ensures good robustness properties as $M_S = 1.03$. This results in the controller parameters $k_p = 4.0$ and $k_i = 7.2$, the adaptive PI controller equation becoming

$$C_O(s) = \frac{0.05}{\hat{G}_P(0)}\left(4.0 + \frac{7.2}{s}\right). \tag{14}$$

The derived controller is fairly standard. However, in our opinion this is only an advantage made possible by the cascaded structure. Using such a simple controller allows us not to waste computational power, that the application could use to serve user requests.

3.5 Design of outer loop feedforward controller

Testing the feedback controller of Section 3.4, we have experienced the need for a better disturbance rejection mechanism for the outer loop. We achieve this with the design of a standard feedforward controller.

Equation (10) shows the outer open loop process dynamics $G_P(s)$. Selecting $\tau_{95}^o = \bar{\tau}_{95}^o$, as well as considering the dynamics in (10) in stationarity, leads to a proposed static feedforward scheme

$$r_q^{ff} = \frac{1}{\hat{G}_I(0)}\frac{\hat{\lambda}}{\hat{\alpha}}\bar{\tau}_{95}^o. \tag{15}$$

Here $\hat{G}_I(0)$, $\hat{\lambda}$ and $\hat{\alpha}$ are estimated as described in Section 4. The feedforward scheme (15) is combined with the feedback controller designed in the previous section, resulting in the complete control structure shown in Figure 5.

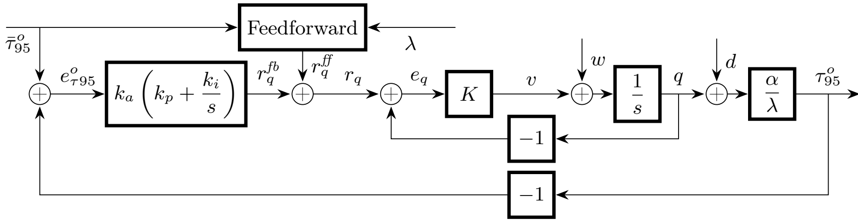


Figure 5. The complete cascaded structure, with the proposed models and controllers for both outer and inner loop.

4. Evaluation

This section presents our results. We validate our control strategy using the open source Python-based brownout simulator[2], built to mimic the behavior of cloud applications [Durango et al., 2014] and described in the following Section 4.1.

4.1 The simulator

The simulator defines the concepts of *Client*, *Request*, *Replica* – a single server, running a brownout application – and *Replica Controller*. Clients issue requests to be served by the replica (server). Clients can behave according to the open-loop or to the closed-loop client model [Schroeder et al., 2006; Alomari and Menasce, 2014]. In the closed-loop model, clients wait for a response and issue a new request only after some think time. In the open loop model, clients do not wait and instead issue new requests with a specific request rate. Being better at modelling a large number of independent users, we performed the evaluation with open-loop clients.

For each request, the simulator computes the service time. The time it takes to serve requests with only the mandatory or with the optional content in addition to the mandatory one are computed as random variables, with normal distributions, whose mean and variance are based on profiling data from the execution of experiments on a real machine [Klein et al., 2014]. The processing time for a request with optional content is a random variable $Y \sim \mathcal{N}(0.07, 0.01)$, while the processing time for the mandatory content is a random variable $Z \sim \mathcal{N}(0.001, 0.001)$. Furthermore, the simulator supports the CFPS queuing discipline with any M_C.

Finally, replicas implement a replica controller, that takes care of selecting – for each request – when to serve optional content. In the simulator, we implemented our own replica controller, described in Section 3. The controller code developed in the simulator can be directly plugged into brownout-aware

[2] https://github.com/cloud-control/brownout-lb-simulator

applications like RUBiS[3] and RUBBoS[4]. For the controller implementation, the adaptive PI controller in (14) was discretized with sample period $h = 0.5$ s using the method suggested in [Åström and Wittenmark, 1997], and complemented by a tracking-based anti-windup solution. The parameter estimations that the feedback and feedforward schemes require ($\hat{G}_P(0)$, $\hat{G}_I(0)$, $\hat{\lambda}$, $\hat{\alpha}$) are implemented as exponentially weighted moving averages according to

$$\hat{y}(k+1) = \beta\hat{y}(k) + (1-\beta)\, y(k). \tag{16}$$

Here $\hat{y}_k$ is the estimate of y, y_k the measurement at time k and $0 \leq \beta \leq 1$ a design parameter. In our simulations we use slightly different β values for the different parameters that we estimate, but mostly $\beta \simeq 0.9$.

4.2 Control validation

The response time requirements of the application are expressed in the form of a maximum value for the 95th percentile of the response times. To bound this value, the controller should be able to constrain the 95th percentile of the response times for the requests that are served with optional content, τ_{95}^o. The remainder of this evaluation focuses on τ_{95}^o, and uses a setpoint $\bar{\tau}_{95}^o = 1$ s.

The adaptive PI controller (denoted by C_{fb}) derived in Section 3.4 is compared with the combined feedback+feedforward scheme (denoted by C_{ff}) from Section 3.5, as well as with the original brownout design (denoted by C_{orig}) described in Section 2.1 and the event-based design[5] (denoted by C_{event} and described in Section 2.2). Since no clear tuning rules were proposed in [Desmeurs et al., 2015], we have tuned its outer controller in the same way as C_{fb} without the adaptive gain. As anticipated, the simulations are performed with Poisson arrivals generated by open-loop clients, and with both $M_C = 3$ and $M_C = 10$, respectively representing behaviors close to FIFO and PS.

Figures 6 to 11 show a simulated sequence (repeated 20 times for statistical significance) of varying arrival rates for both values of M_C. The arrival rates vary following the sequence $\{20, 100, 30, 70, 20\}$ s^{-1}, representing step changes in the load d, and each value is kept constant for 60 s. Figures 6 to 9 show the 95% confidence intervals of the plotted quantities. The upper plots show the derivative of the queue length (i.e., the $v = \dot{q}$ control signal), displaying both the computed (v) and the actuated control signal (v_{actual}). The middle plots show the actual queue length q and its reference value r_q (i.e., the outer loop control signal). Finally, the bottom plots display the response times τ_{95}^o and its setpoint $\bar{\tau}_{95}^o = 1$. Figures 10 and 11 show a comparison of all

[3] https://github.com/cloud-control/brownout-rubis
[4] https://github.com/cloud-control/brownout-rubbos
[5] In the event-based design, we use τ_{95}^o as measurement signal, for fairness with respect to our solution.

the four strategies. The upper plots represent the dimmer value θ (i.e., the percentage of requests served with optional content), which is determined by the controller in the case of C_{orig} and *a posteriori* computed in the case of the other strategies. The middle plots show the reference values of the queue length r_q for the proposed strategies (C_{fb}, C_{ff}) and for the event-based controller (C_{event}). The lower plots show τ^o_{95} and its setpoint $\bar{\tau}^o_{95} = 1$. The plots of Figures 10 and 11 show average values over the 20 repeated sequences for readability.

Figures 6 to 9 show one of the benefits of a cascaded structure: the inner loop can be very fast[6], allowing a tighter control. In some cases (e.g., Figures 7–9, in the time interval 60 s–120 s) the system experiences some actuation errors, leading to a stationary error in the inner loop. Thanks to the integral action in the outer controller, response times τ^o_{95} are still kept close to their setpoints. In fact, the inner loop is in general able to follow the outer loop control signal r_q and drive the queue length q to acceptable values. The good control performance that we experience can be linked directly to the model being a better approximation compared to previous models [Klein et al., 2014]. The cascaded structure C_{fb} shows no overshoot in the queue setpoints but is slower in responding to changes in d, while C_{ff} is faster in handling changes in the arrival rates, but overshoots. For $M_C = 10$, the C_{ff} controller gets larger overshoots in its outer loop control signal r_q, as a result of the assumed model (10) not describing the dynamics as well as for $M_C = 3$. However, this has a minimal effort on the control performance.

Looking at Figures 10 and 11, the amount of optional content served (shown in the θ plot) is an indication of how well the application behaves in terms of potential revenues for the application owner. Both in the case of the event-based strategy C_{event} and our proposals C_{fb} and C_{ff}, the average dimmer value is 28%, while the original strategy C_{orig} only achieves an average value of 25% optional content served. Quite naturally, as the arrival rate increases the amount of optional content served decreases. As a result of the robust design, the control performances of C_{fb} and C_{ff} are able to serve additional optional content, while keeping the response times around the setpoint under the different conditions, clearly outperforming the original design. Note that the results are truncated for C_{orig}, its peak values of τ^o_{95} reaches about 5 seconds for both M_C.

Table 1 presents quantitative data comparing the four strategies in the same 20 repeated simulations, for both values of M_C. The first two columns show the Integral of the Absolute Error ($\int |e^o_{\tau 95}(t)| dt$), the following columns show the variance of *all* the optional content response times τ^o and the last two columns show the maximum value of τ^o. The developed controllers are very close to the event-based controller [Desmeurs et al., 2015], but provide

[6] As there is no physical actuator involved, the aggressive behavior is not an issue.

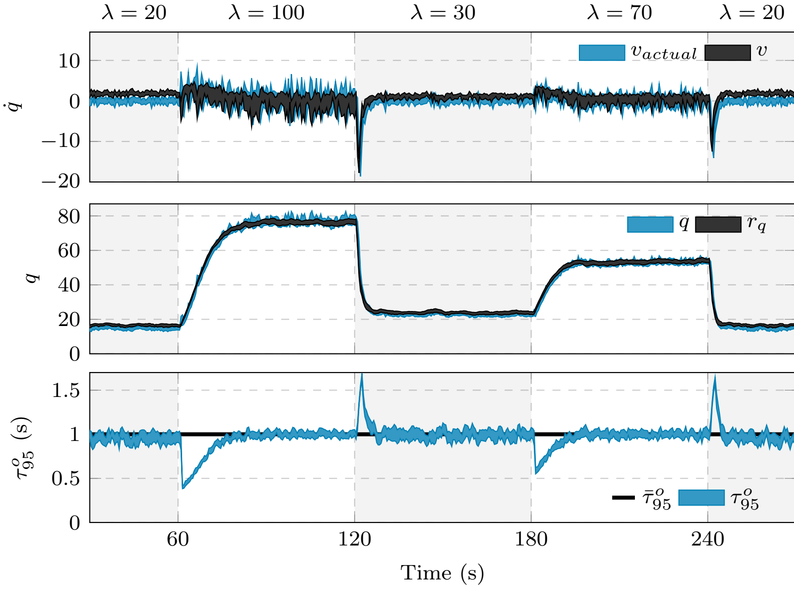


Figure 6. 95% confidence interval plots for C_{fb} with $M_C = 3$.

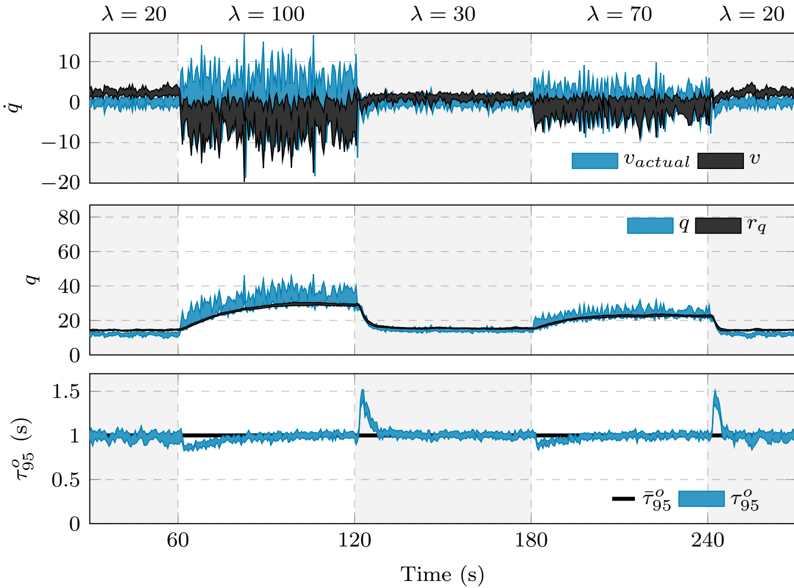


Figure 7. 95% confidence interval plots for C_{fb} with $M_C = 10$.

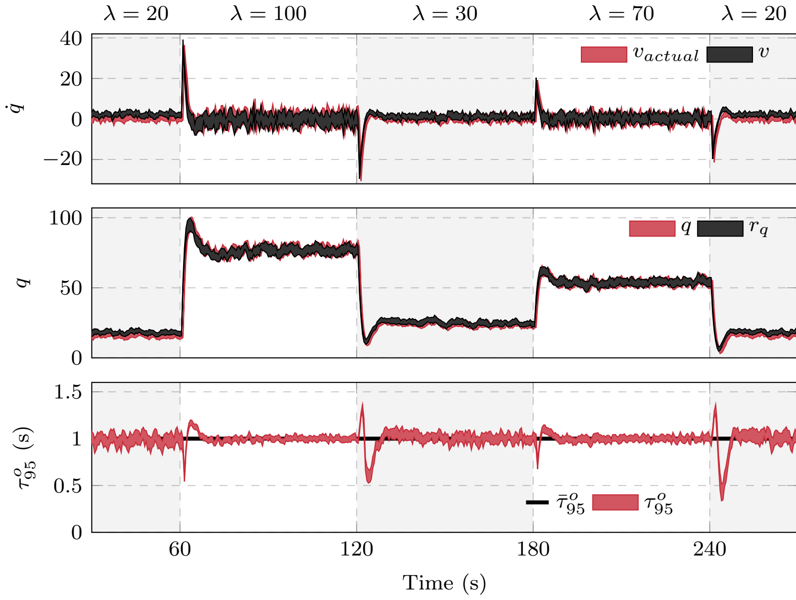


Figure 8. 95% confidence interval plots for C_{ff} with $M_C = 3$.

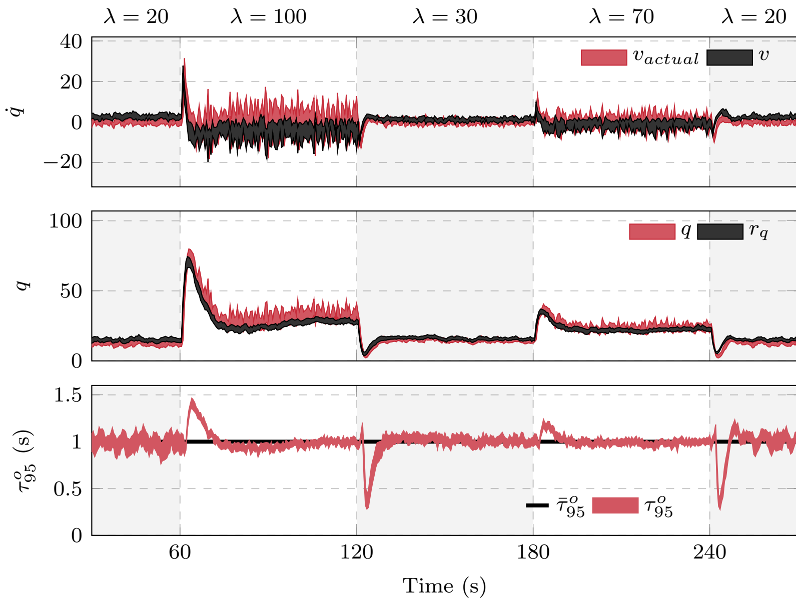


Figure 9. 95% confidence interval plots for C_{ff} with $M_C = 10$.

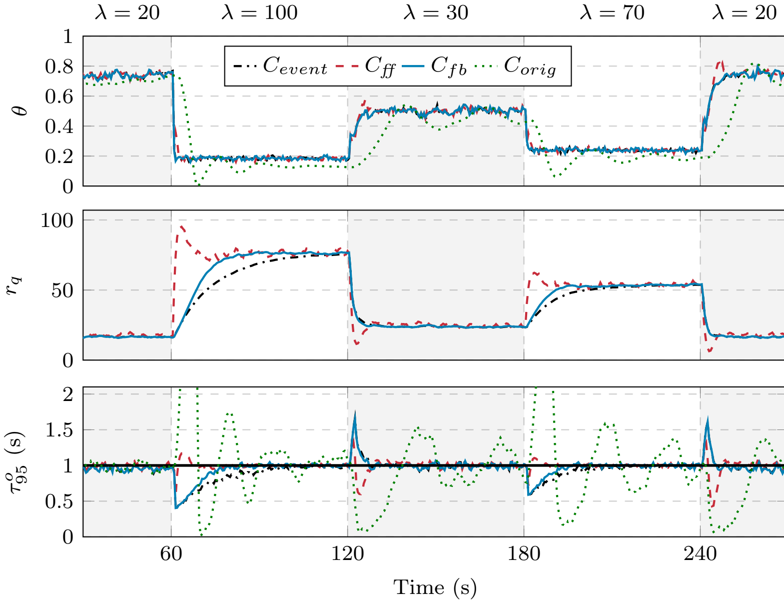


Figure 10. Average value plots for all 4 strategies with $M_C = 3$.

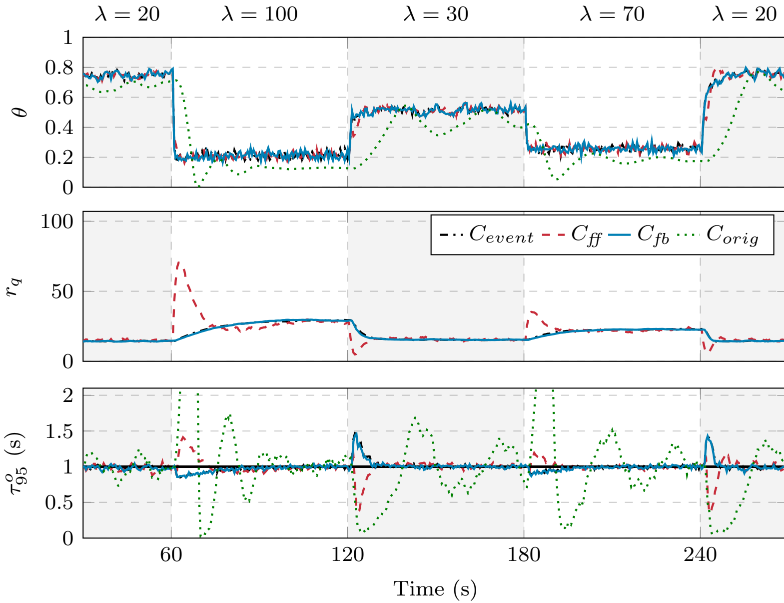


Figure 11. Average value plots for all 4 strategies with $M_C = 10$.

Table 1. Quantitative comparison of all 4 strategies.

	IAE [$\cdot 10^3$s]		var(τ^o) [s]		τ^o_{max} [s]	
M_C	3	10	3	10	3	10
C_{orig}	8.23	8.34	0.695	0.745	5.68	6.27
C_{event}	1.58	1.01	0.031	0.021	1.82	1.93
C_{fb}	1.48	0.98	0.030	0.021	1.81	1.83
C_{ff}	1.23	1.43	0.026	0.034	1.56	1.65

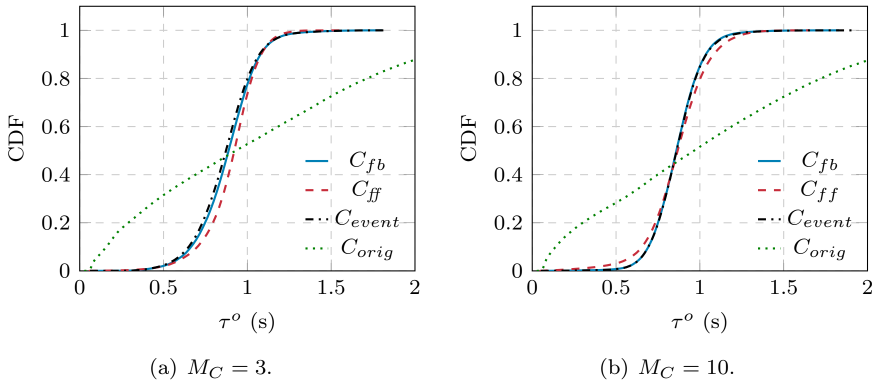


Figure 12. Empirical cumulative distributions of τ^o for all 4 strategies.

formal guarantees. Also, especially with C_{ff} the maximum response time is lower than with C_{event}.

Finally, to complete our evaluation, we computed the empirical Cumulative Distribution Function (CDF) of τ^o for both values of M_C, using the four control strategies. Also in this case, the simulations are repeated 20 times (but not averaged). Figure 12 shows the results, indicating clearly that the control strategies synthesized in this paper outperform the original design [Klein et al., 2014] C_{orig}, and behave similarly to the event based controller C_{event}. C_{ff} and C_{fb} display much shorter tails in the response times, and are able to keep the 95$^{\text{th}}$ percentile close to 1 second. Finally, C_{ff} is able to keep the tails slightly shorter than C_{fb}, thanks to its faster reactions to changes in the arrival rate.

5. Conclusion and future work

In this paper a novel brownout controller was presented, capable of combining the benefits of both the event-based brownout [Desmeurs et al., 2015] in terms of performance and the advantages of the original approach [Klein et

al., 2014], in terms of analysis.

This research was motivated by the desire of solving the autoscaling problem for brownout applications – i.e., to decide when to start a new virtual machine for the same cloud application, taking also advantage of the knowledge of the dimmer value and not only of the response times. We have realized that the brownout loop, in any of its forms, was not suitable for being directly extended with autoscaling capabilities and there was a need for a more realistic model of the behavior of the application. Together with a better control strategy, this paper provides such a model, which we plan to use for brownout-aware autoscaling.

Acknowledgments

This work was partially supported by the Wallenberg Autonomous Systems and Software Program (WASP), by the Swedish Research Council (VR) for the projects "Feedback Computing" and "Power and temperature control for large-scale computing infrastructures", by the LCCC Linnaeus Center and, by the ELLIIT Excellence Center at Lund University.

References

Abdelzaher, T. F., K. G. Shin, and N. Bhatti (2002). "Performance guarantees for web server end-systems: A control-theoretical approach". *IEEE Transactions on Parallel and Distributed Systems* **13**:1. ISSN: 1045-9219.

Alomari, F. and D. A. Menasce (2014). "Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks". *IEEE Trans. Parallel Distrib. Syst.* **25**:6. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.70. URL: http://dx.doi.org/10.1109/TPDS.2013.70.

Åström, K. J. and B. Wittenmark (1997). *Computer-controlled Systems (3rd Ed.)* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN: 0-13-314899-8.

Barna, C., M. Fokaefs, M. Litoiu, M. Shtern, and J. Wigglesworth (2016). "Cloud adaptation with control theory in industrial clouds". In: *2016 IEEE International Conference on Cloud Engineering Workshop, IC2E Workshops, Berlin, Germany, April 4-8, 2016*. DOI: 10.1109/IC2EW.2016.13. URL: http://dx.doi.org/10.1109/IC2EW.2016.13.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an m/g/1/k* ps queue". In: *Telecommunications, 2003. ICT 2003. 10th International Conference on*. Vol. 2. IEEE.

Dean, J. and L. A. Barroso (2013). "The tail at scale". *Commun. ACM* **56**:2. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. URL: http://doi.acm.org/10.1145/2408776.2408794.

Deliparaschos, K. M., T. Charalambous, E. Kalyvianaki, and C. Makarounas (2016). "On the use of fuzzy logic controllers to comply with virtualized application demands in the cloud". In: *2016 European Control Conference, ECC 2016, Aalborg, Denmark, June 29 - July 1, 2016*. DOI: 10.1109/ECC.2016.7810362. URL: http://dx.doi.org/10.1109/ECC.2016.7810362.

Desmeurs, D., C. Klein, A. V. Papadopoulos, and J. Tordsson (2015). "Event-driven application brownout: reconciling high utilization and low tail response times". In: *2015 International Conference on Cloud and Autonomic Computing*. DOI: 10.1109/ICCAC.2015.25.

Ding, S., S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas (2011). "Indexing strategies for graceful degradation of search quality". In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '11. ACM, Beijing, China. ISBN: 978-1-4503-0757-4. DOI: 10.1145/2009916.2009994. URL: http://doi.acm.org/10.1145/2009916.2009994.

Durango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with brownout". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. DOI: 10.1109/CDC.2014.7040221. URL: http://dx.doi.org/10.1109/CDC.2014.7040221.

Gupta, V., M. H. Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2004). *Feedback Control of Computing Systems*. John Wiley & Sons. ISBN: 047126637X.

Hoflack, L., S. Vuyst, S. Wittevrongel, and H. Bruneel (2008). "Modeling web server traffic with session-based arrival streams". In: *Proceedings of the 15th International Conference on Analytical and Stochastic Modeling Techniques and Applications*. ASMTA '08. Springer-Verlag, Nicosia, Cyprus. ISBN: 978-3-540-68980-5. DOI: 10.1007/978-3-540-68982-9_4. URL: http://dx.doi.org/10.1007/978-3-540-68982-9_4.

Jalaparti, V., P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan (2013). "Speeding up distributed request-response workflows". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. ACM, Hong Kong, China. ISBN: 978-1-4503-2056-6.

Kalyvianaki, E., T. Charalambous, and S. Hand (2014). "Adaptive resource provisioning for virtualized servers using kalman filters". *TAAS* **9**:2. DOI: 10.1145/2626290. URL: http://doi.acm.org/10.1145/2626290.

Khalil, H. K. (1996). "Noninear systems". *Prentice-Hall, New Jersey* **2**:5.

Kjaer, M. A., M. Kihl, and A. Robertsson (2007). "Response-time control of a single server queue". In: *2007 46th IEEE Conference on Decision and Control*. DOI: 10.1109/CDC.2007.4434617.

Kjaer, M. A., M. Kihl, and A. Robertsson (2009). "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers". *IEEE Transactions on Network and Service Management* **6**:4. ISSN: 1932-4537.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: building more robust cloud applications". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. ACM, Hyderabad, India. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568227. URL: http://doi.acm.org/10.1145/2568225.2568227.

Kleinrock, L. (1975). *Queueing Systems*. Vol. I: Theory. Wiley Interscience.

Lu, C., J. A. Stankovic, S. H. Son, and G. Tao (2002). "Feedback control real-time scheduling: framework, modeling, and algorithms*". *Real-Time Systems* **23**:1. ISSN: 1573-1383. DOI: 10.1023/A:1015398403337. URL: http://dx.doi.org/10.1023/A:1015398403337.

Maggio, M., C. Klein, and K.-E. Årzén (2014). "Control strategies for predictable brownouts in cloud computing". *IFAC Proceedings Volumes* **47**:3. 19th IFAC World Congress. ISSN: 1474-6670.

Mars, J., L. Tang, R. Hundt, K. Skadron, and M. L. Soffa (2011). "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. ACM, Porto Alegre, Brazil. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155650. URL: http://doi.acm.org/10.1145/2155620.2155650.

Schroeder, B., A. Wierman, and M. Harchol-Balter (2006). "Open versus closed: a cautionary tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. USENIX Association, San Jose, CA. URL: http://dl.acm.org/citation.cfm?id=1267680.1267698.

Yun, S.-Y. and A. Proutiere (2015). "Distributed proportional fair load balancing in heterogenous systems". In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '15. ACM, Portland, Oregon, USA.

ISBN: 978-1-4503-3486-0. DOI: 10.1145/2745844.2745861. URL: http://doi.acm.org/10.1145/2745844.2745861.

Paper II

Improved Dynamic Modeling for Controlled Server Queues

Tommi Berner **Johan Ruuskanen** **Martina Maggio**
Karl-Erik Årzén

Abstract

Resource provisioning for applications hosted in the cloud is a difficult task due to inherent performance variabilities in the infrastructure. Control theory has proven to be an efficient tool for dynamic resource provisioning, increasing the predictability of the applications. However, an important prerequisite for a successful control design is an adequate model of the dynamics involved in the cloud. In this paper we focus on modeling of *controlled* server queues that are subject to actuators, such as frequency scaling or admission control. We show that the models of today are only applicable to very specific server types, characterized by their *queuing disciplines*, and propose a model structure that can be applied under more general settings. Our structure is nonlinear, yet simple enough to allow for control design. We further describe how our model structure can be adapted to the dynamics of a server under the general limited processor sharing discipline. We compare our approach to state-of-the-art models in an extensive simulation campaign, showing the superior versatility of our model structure. Finally, we use our model in a control design example to show the insights that can be gained from our proposed structure. We identify a critical frequency range where the characteristics of the involved service time distribution affects the control design, and where a more advanced controller structure might be needed. We also use our model structure to identify safe frequency ranges where simpler controllers can be utilized regardless of service time distribution.

Journal Manuscript under Review.

1. Introduction

The motivation for applying control theory to the computing systems domain has increased with the growth in popularity of the cloud computing paradigm, where uncertainty (and, hence, the potential gain when applying control techniques) is an inherent component of the computing infrastructure [Barroso and Hölzle, 2009]. Applications deployed in the cloud typically run on virtual machines or containers, which can exhibit considerable variations in their performance, due to e.g. resource contention and hardware heterogeneity. In this uncertain environment and due to the wide performance range, guaranteeing predictable application behavior (e.g. availability and responsiveness) is a difficult task. One common remedy to this problem is resource over-provisioning [Barroso and Hölzle, 2009], where the cloud deployment is statically designed with respect to heavy load scenarios. This is, however, a costly approach that lowers the resource usage. A technique that has been applied successfully to improve the resource usage is the use of feedback control [Pothukuchi et al., 2020; Abdelzaher et al., 2003; Abdelzaher and Chenyang Lu, 2000; Patikirikorala et al., 2012; Klein et al., 2014]. Using control allows us to create a dynamic strategy for resource allocation, that follows the current user demand.

Feedback control can be used to regulate the amount of requests queued in a server. Controlling the request queues is an efficient way to guarantee predictable application response times, as actions are taken if the queues grow too large. The key for designing a fast and reliable queue length control lies in the model, that needs to be able to capture the relevant dynamics present in the system, while not being overly complex.

In this paper, we focus on dynamic modeling of server queues. We consider the requests entering the servers as a stream, rather than packet-by-packet. This dynamic modeling approach is known as *fluid* [Kulkarni, 1997]. For simplicity, we will omit the usage of the word fluid in this paper, and only denote the models as *dynamic*.

Today, dynamic models for server queues exist both in the field of queuing theory [Wang et al., 1996; Perez and Casale, 2017], and in the field of control theory [Arcelli et al., 2015; Paganini et al., 2012]. In queuing theory, the vast majority of the models concern servers that are *not* subject to any actuators. These are denoted as *non-controlled* servers, to differentiate them from *controlled* servers [Kitaev and Rykov, 1995] that are subject to actuation. Examples of possible server actuators include: frequency scaling [Kim et al., 2015], admission control [Kihl et al., 2004], and graceful degradation techniques [Klein et al., 2014; Nylander et al., 2018].

The modeling approach presented in this paper is intended for controlled servers. An important characteristic that distinguishes this type of server from non-controlled ones is that the dynamic models for the controlled servers

always assume that the queues are non-empty. Naturally, the model from the control signal to the output signal is only relevant if the queue (that can be actuated upon) contains requests. The most commonly used model today for this controlled queue setting is a simple integrator (see e.g. [Arcelli et al., 2015; Nylander et al., 2018]), which is sometimes not accurate enough. As shown later in this paper, the integrator model accuracy depends on the processing behavior of the servers, characterized by the queuing discipline. In our paper, we go beyond the two most common disciplines Processor Sharing (PS) and First Come, First Served (FCFS), and explore how the queue length dynamics are affected by a generalization of the two disciplines, denoted as Limited Processor Sharing (LPS) [Zhang et al., 2009] and further described in Section 2.

Paper contribution. In this paper:

- We introduce the state-of-the-art server queue models, and argue that for some types of servers the integrator model is insufficient. This depends on the servers' processing behaviors, characterized by the *queuing discipline*.
- We propose a non-linear, yet simple, model structure that is suitable for control design under the more general queuing discipline LPS. The model is motivated by the need and trade-off between simplicity, efficiency, and accuracy.
- We evaluate our model structure using a discrete-event simulator, and compare it to existing approaches. Our evaluation shows that our model is the most versatile one, and performs well for a wider range of queuing disciplines.
- We use our model structure for a control design example under the PS discipline to identify safe and critical frequency regions. In the critical region the characteristics of the service time distribution affect the control design complexity, whereas the safe regions allow for a simpler control design regardless of service time distribution.

To the best of our knowledge, this is the first research contribution that considers the processing behaviour as a first class citizen in the modelling phase for controlled queues, obtaining thereby a model that is versatile enough to handle multiple queuing disciplines (and in particular LPS). The rest of the paper is organized as follows. Section 2 introduces relevant concepts from queuing theory, and describes related dynamic server queue models. In Section 3, we propose our own model, motivated by a simulation study. We evaluate our model structure in Section 4 and use our model in a control design example in Section 5. We evaluate the design example in Section 6 and Section 7 concludes the paper.

2. Background and Related Work

This section introduces relevant concepts from queuing theory that will be used in the paper. We also describe related work, consisting of existing dynamic server queue models, for both non-controlled and controlled servers.

2.1 Queuing theory concepts

The field of queuing theory [Kleinrock, 1975] provides models for servers that handle user requests. These servers are modeled as *queues*. Queue models are based on statistics and their main usage is to determine server utilization and mean response times, the latter is used to check that the server satisfies basic user requirements and the former is used to ensure that the server is providing useful work.

The arrival of server users (or clients) is captured by a random variable Y following an inter-arrival time distribution with rate $\lambda = 1/\mathbf{E}(Y)$. The behavior of the server (including the request sizes) is represented by a random variable X following a service time distribution with mean $\bar{x} = \mathbf{E}(X)$. The distributions of X and Y can be fully described by either a Cumulative Distribution Function (CDF), $F(z) = P(Z \leq z)$, or a Probability Density Function (PDF) $f(z) = d/dz\, F(z)$, with z a general stochastic variable.

The processing of a request is modeled using queuing disciplines [Kleinrock, 1975]. One of the most commonly used disciplines (from the theoretical perspective) is First Come, First Served (FCFS). A queue under FCFS processes only one request at a time (rather than more than one simultaneously). Any other request (that arrived after the one that is currently served) waits in a queue of infinite size and is then served in the arriving order. Another commonly used discipline is Processor Sharing (PS). Using PS, all requests are processed simultaneously, and each request gets a share of the computing processor that is equal to the reciprocal of the number of requests that are currently served. The PS discipline is normally a good approximation of the real behavior of servers, as it can be viewed as an idealization of a time-sharing scheduling protocol. However, processing too many requests simultaneously can lead to considerable overhead due to switching [Zhang et al., 2009].

A natural generalization of the FCFS and PS disciplines is to only allow for a maximum of M_C requests to be processed simultaneously, resembling the multi-threaded structure of common applications and servers. While these requests are processed, the others wait in a queue. As soon as one of the currently processed requests terminates, the next in line is picked from the queue and served. This generalized queuing discipline is denoted as *Limited* Processor Sharing (LPS). Notice that both FCFS and PS are special cases of LPS. FCFS is equivalent to LPS with $M_C = 1$ and PS is equivalent to LPS with $M_C = \infty$ [Zhang et al., 2009].

2.2 Dynamic models for non-controlled queues

The queuing theory models for server queues are often *not* directly applicable to the field of automatic control, as they mostly describe average behaviors for servers that are not subject to any actuators, i.e. non-controlled. To get models that converge for average values over time, a very common prerequisite in queuing theory is then to demand that the service rate $\mu = 1/\bar{x}$ is greater than the arrival rate λ, i.e. $\lambda < \mu$ [Kleinrock, 1975].

Under the assumption $\lambda < \mu$, there exists dynamic models for non-controlled servers, such as the pointwise stationary fluid flow approximation (PSFFA) [Wang et al., 1996], which includes support for many service time distributions. Other examples include fluid models for more general phase-type distributions under closed queuing networks [Perez and Casale, 2017; Schwarz et al., 2016].

One important exception is an asymptotic dynamic model for non-controlled PS queues [Jean-Marie and Robert, 1994]. It investigates the case $\lambda > \mu$, i.e. queues that grow to infinity. As this implies that the queues will always be non-empty, the results are actually highly applicable to control applications even without the presence of actuators. The authors of [Jean-Marie and Robert, 1994] find that the asymptotic growth rate α of PS queues is actually different from FCFS queues. Their main result for PS queues states that $\alpha > 0$ is the solution to the following integral equation:

$$\alpha = \lambda \left(1 - \int_0^\infty e^{-\alpha x} f(x)\, dx \right). \tag{1}$$

The equation above involves the complete PDF $f(x)$ of the service time distribution, which implies that, for PS queues, α is dependent on the *entire* distribution, and not only its mean $\bar{x}$. This is however not the case for the inter-arrival distribution, as α is only affected by its rate λ and not the shape of its distribution. The well-known corresponding, much simpler, result for the asymptotic growth β in FCFS queues is also given in [Jean-Marie and Robert, 1994]:

$$\beta = \lambda - \frac{1}{\bar{x}}. \tag{2}$$

For FCFS queues, the growth rate is thus only dependent on arrival rate λ and the mean service time $\bar{x}$, regardless of the distributions.

Another important parameter in dynamic queue models is the remaining work w, defined as the sum of the remaining service requirements, w_i, for all n requests in the server:

$$w(t) = \sum_{i=1}^{n} w_i(t). \tag{3}$$

The dynamics for the remaining work w is a well-known result in queuing theory, given in e.g. [Jean-Marie and Robert, 1994]:

$$\dot{w} = \lambda \bar{x} - 1, \tag{4}$$

that holds for *any* work-conserving queuing discipline, including both FCFS, PS and LPS [Jean-Marie and Robert, 1994].

2.3 Dynamic models for controlled queues

The current state-of-the-art dynamic queue model for controlled servers is very simple. It utilizes the fundamental physics involved in a queue (from e.g. [Arcelli et al., 2015]):

$$\dot{q} = r_i(t) - r_o(t), \tag{5}$$

with q the queue length and r_i and r_o the input and output rate of requests. For non-empty queues, i.e. $q(t) > 0$, the relation (5) is exact for all involved queuing disciplines in this paper. However, the most common assumption made when turning (5) into a model from some normalized control input $0 < u \leq 1$ to queue length q, is that there exists a static relation between r_o and u. This results in the following model (see e.g. [Arcelli et al., 2015; Nylander et al., 2018]):

$$\dot{q} = \lambda - \frac{1}{\bar{x}} u. \tag{6}$$

Comparing (6) to the queuing theory results presented earlier in this section, we see that it perfectly matches the asymptotic growth rate β for FCFS queues, stated in Equation (2). However, it does *not* match the corresponding asymptotic growth rate for PS, denoted as α from Equation (1). α is clearly dependent on the entire service time distribution, whereas the dynamics in (6) only considers the mean. It can thus be expected that the state-of-the-art model describes the queue dynamics poorly for servers with a processing behavior closer to PS than FCFS.

For network bandwidth applications, a dynamic model has been proposed to describe a proportional bandwidth allocation protocol [Paganini et al., 2012]. While the authors Paganini et.al. do not use it for queue length control, it can be easily seen that proportional bandwidth sharing is analogous to PS queue length dynamics as they both share processor capacity in the same way. The model that they propose is the following partial differential equation (PDE):

$$\frac{\partial q(t,x)}{\partial t} = \frac{\partial q(t,x)}{\partial x} \frac{u(t)}{q(t)} + \lambda \bar{F}(x), \tag{7}$$

with $\bar{F}(x)$ the complementary CDF of service times, i.e. $\bar{F}(x) = 1 - F(x)$. In our paper we will denote this model (7) as the Paganini model $\mathcal{M}_P$.

Evaluating (7), it can be seen that its asymptotic behavior perfectly matches the growth rate α for PS queues. Furthermore, the authors of [Paganini et al., 2012] formally show that the model (7) indeed perfectly describes the *complete* dynamics of the proportional bandwidth sharing protocol, and thus also the PS queue dynamics. While the Paganini model is exact for PS queues, it is also very difficult to use for control design due to its PDE structure. To the best of our knowledge, this model has not been used so far to control PS queue lengths, most likely due to its complexity. Hence, there is a need for a simpler more control-oriented model.

3. Model

This section motivates and describes our proposed model structure for controlled server queues. We assume that the queues are always non-empty, which is a reasonable assumption for a server control scenario. The main actuator that we will focus on in this paper is *server speed*, denoted as $0 < u \leq 1$, represented by e.g. changes to a normalized CPU frequency. Furthermore, our model structure is designed to fit the behavior of PS queues. We will later in this section show that our model structure can handle both other actuators and be applied to the generalized queuing discipline LPS.

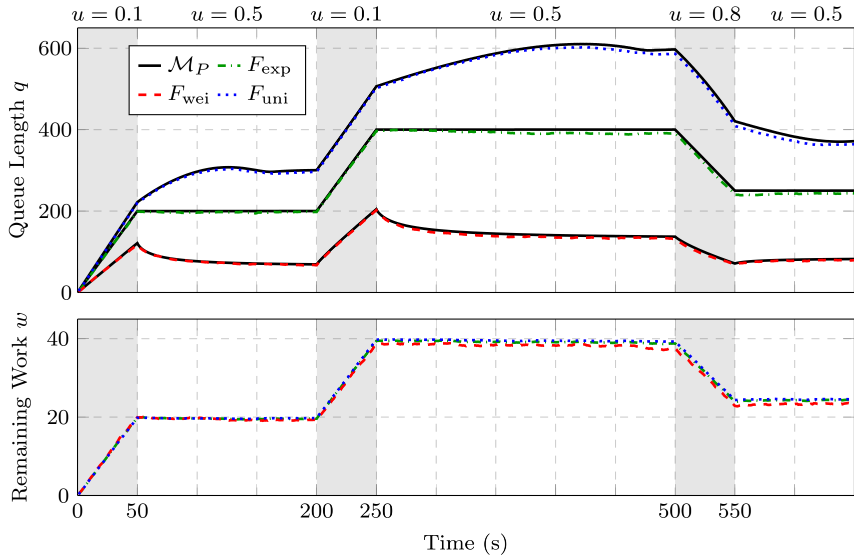
3.1 Simulation study

We motivate our structure based on known results from queuing theory, physical relations and the behaviors observed in an initial simulation study. Our simulations are performed using a discrete-event simulator developed in Python, described in Section 4.1.

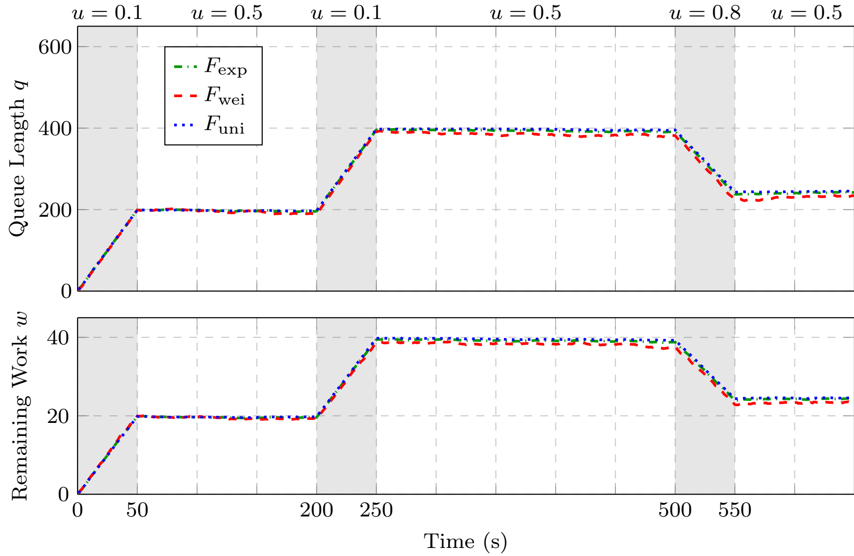
The simulation study is performed in open loop, for both queuing disciplines PS and FCFS, and for three example service time distributions with same mean $E(x) = 0.1$s:

- Exponential $F_{\text{exp}}(x) = 1 - e^{-10x}, \quad x \geq 0$
- Uniform $F_{\text{uni}}(x) = \begin{cases} x/0.2 & \text{if } 0 \leq x \leq 0.2 \\ 1 & \text{if } x > 0.2 \end{cases}$
- Weibull $F_{\text{wei}}(x) = 1 - e^{-(20\,x)^{0.5}}, \quad x \geq 0$

For the arrival of new requests we use the Poisson process with rate $\lambda = 5/\text{s}$, however, as shown in [Paganini et al., 2012] the dynamics for both PS and FCFS are *not* affected by the arrival distribution except for its mean. The control input u, representing server speed, is changed in steps as the sequence u_{steps} ={0.1, 0.5, 0.1, 0.5, 0.8, 0.5}. For each service time distribution, the



(a) PS.



(b) FCFS.

Figure 1. Results from the simulation study. The upper plots show queue lengths q and the lower the remaining work w.

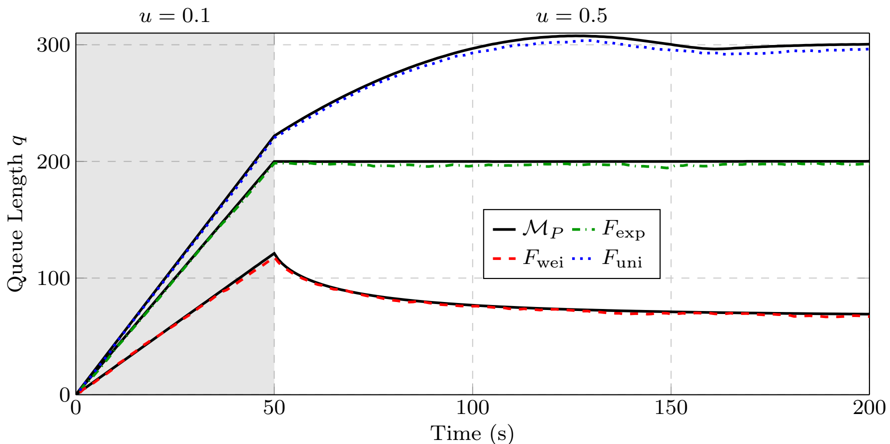


Figure 2. First 200 seconds of the queue length plot in Figure 1(a).

test is repeated 50 times with different random seeds. The average of all 50 sequences is then formed to ensure that we catch the statistical behaviors.

The results of the simulation study are shown in Figure 1, where Figure 1(a) shows the queuing discipline PS and Figure 1(b) the results for FCFS. The figures show both the queue length dynamics and the remaining work w. As expected, since all three service time distributions have the same mean value, their queue length behaviors for FCFS are identical, acting like pure integrators. The remaining work dynamics also comply with the theory from Section 2, stating that FCFS and PS should have the exact same, pure integrator, behavior. The most interesting plot is, however, the one showing the queue length dynamics for PS. Here we see that the three example distributions act completely differently, despite having the same mean value $E(x)$. The exponential distribution acts like a pure integrator, whereas Weibull and uniform exhibit more complex dynamics. In fact, they suggest a nonlinear behavior as the settling time after each step in u changes with the level of w. The queue length simulation behaviors for PS are plotted together with the theoretical Paganini model $\mathcal{M}_P$ results for comparison. Our simulations match the theoretical model results for PS very closely, confirming the accuracy of our discrete-event simulator.

As a first step to describe the PS queue length dynamics, we consider the first 200s of Figure 1(a), consisting of a single step in u, seen in greater detail in Figure 2. Using the System Identification Toolbox[1] in Matlab, we use the simulation input-output data of Figure 2 to identify linear continuous-time, transfer function models. We compare five different pole-zero configurations

[1] `https://www.mathworks.com/products/sysid.html`

Table 1. Linear identification experiment data.

		Configuration (n_p,n_z)				
		1,0	1,1	2,1	2,2	3,2
F_{uni}	Fit (%)	64.3	72.3	**95.8**	95.9	96.4
	MSE	981	588	**13.8**	12.8	10.2
F_{exp}	Fit (%)	**94.2**	94.2	94.3	94.4	95.1
	MSE	**9.92**	9.92	9.50	9.24	7.02
F_{wei}	Fit (%)	57.1	76.0	**83.2**	83.4	83.8
	MSE	99.2	31.0	**15.3**	14.6	14.2

(n_p, n_z) with number of poles n_p ranging from 1 to 3 and zeros n_z from 0 to 2. The results are seen in Table 1, quantified by the measures model fit and mean squared error (MSE). The values in bold highlight the simplest configuration with acceptable metrics, hence the best choice without over-fitting. For F_{exp} the best choice is 1,0, whereas for F_{uni} and F_{wei} it is 2,1. All best-fit configurations include an integrator pole, and the 2,1 configurations include an additional stable pole and stable zero. Given the very small differences between the simulation data and Paganini model results seen in Figures 1 and 2, these best-fit configurations also serve as approximate, numerical linearizations of the Paganini model stated in Equation (7). Hence, assuming a common model structure, the data of this identification experiment suggests that our proposed model should linearize to a system of configuration 2,1.

3.2 Model structure

Based on the results of the initial simulation study, the dynamic queue model should thus fulfill a set of requirements. It should:

- Be non-linear (time constants depend on w)
- Contain the integrating state w
- Linearize to a system with an integrator pole, a stable pole and a stable zero
- Reduce to a pure integrator for the exponential distribution

Furthermore, the model structure also has to comply with the fundamental physics of queues:

- Queue growth $\dot{q}$ = input rate $r_i(t)$ − output rate $r_o(t)$

The input rate r_i is trivial to model as the request arrival rate λ. However, what is more interesting is how to relate the output rate r_o to the involved parameters, i.e. remaining work w, queue length q and server speed u. As seen in the simulation study, r_o depends on the service time distribution for PS queues, which implies that a new parameter is necessary to add to the model. We denote this distribution specific parameter as k, which describes the impact on the output rate r_o from the service time distribution. Furthermore, the simulation study shows that the time constant of the pole increases with w, which implies that output rate can be modeled as proportional to the inverse of w. The insights gained from the simulation study, as well as the fundamental physics of queues thus motivate our suggested model structure. The structure, that fulfills all of the above requirements, is the following

$$\begin{aligned} \dot{w} &= \lambda \bar{x} - u \\ \dot{q} &= \lambda - k \frac{q}{w} u, \end{aligned} \tag{8}$$

with $\lambda > 0$, $q > 0$, $\bar{x} > 0$, $w > 0$ and $k > 0$. The equation for $\dot{w}$ in (8) is completely based on previous theory for remaining work dynamics, presented in Section 2. The novelty of our model structure thus lies in the queue length dynamics, using a structure as motivated above.

A linearization of (8) around an operating point (w_0, q_o, u_0) results in the following linear system:

$$\begin{aligned} \Delta\dot{w} &= -\Delta u \\ \Delta\dot{q} &= \frac{kq_0u_0}{w_0^2}\Delta w - \frac{ku_0}{w_0}\Delta q - \frac{kq_0}{w_0}\Delta u, \end{aligned} \tag{9}$$

with $\Delta w = w - w_0$, $\Delta q = q - q_0$ and $\Delta u = u - u_0$. Applying the Laplace transform on (9) gives the following linear transfer function from ΔU to ΔY:

$$G_N(s) = -\frac{q_0}{w_0} \frac{\left(1 + s\frac{w_0}{u_0}\right)}{s\left(1 + s\frac{w_0}{ku_0}\right)}, \tag{10}$$

which fulfills the specification on a linearized model with one integrator pole, a stable zero and a stable pole.

One important property of our proposed model structure is that for $k = 1$, (8) reduces to a linear integrator model:

$$\dot{q} = \lambda - \frac{1}{\bar{x}} u, \tag{11}$$

which is identical to the model for FCFS queues. This can be seen by observing that in (8), with $k = 1$, λ a time-varying continuous function and $\bar{x}$

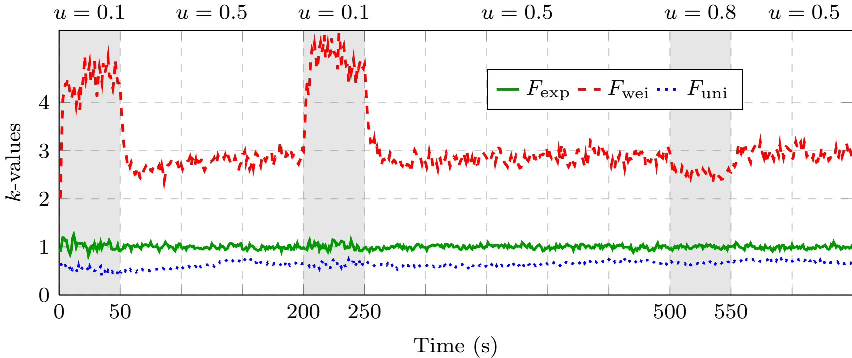


Figure 3. k-values from the simulation study for the PS discipline.

Table 2. Theoretical k_∞-values for the simulation study.

	Theoretical k_∞-values		
	$u = 0.1$	$u = 0.5$	$u = 0.8$
F_{uni}	0.51	0.67	0.74
F_{exp}	1.00	1.00	1.00
F_{wei}	4.25	3.00	2.94

a constant, the states q and w will keep their constant relation $q = w/\bar{x}$ if $q(t_0) = w(t_0)/\bar{x}$ for some initial time t_0. For $k = 1$, (8) can thus be reduced to a first-order linear model with the dynamics given by (11).

The values for k can be estimated online, but it is also possible to use the asymptotic slope α to determine asymptotic values for k, denoted as k_∞. Solving the differential equation (8) with initial conditions $q(0) = w(0) = 0$, and comparing the resulting slope with Equation (1) for α, we get the following relation

$$k_\infty = \frac{\lambda\,(\lambda\,\bar{x} - \alpha\,\bar{x} - u) + \alpha\,u}{\alpha\,u}, \tag{12}$$

which is valid for any α. However, the integral equation in (1) for determining α is only strictly valid for $\alpha > 0$. In our simulations, we have noticed that Equation (1) calculates accurate negative slopes $\alpha < 0$ as well, as long as the integral converges. As we have not been able to mathematically prove the correctness of the negative slopes, they only serve as indicative values of k_∞.

The k-values for the three example distributions from the simulation study are shown in Figure 3. As can be seen, the behavior of F_{wei} corresponds to $k > 1$, F_{exp} to $k = 1$ and F_{uni} to $k < 1$. There are some variations within the distributions, especially for F_{wei}, triggered by scenario changes.

For comparison with the simulation results from Figure 3, the theoretical k_∞-values can be seen in Table 2. The k_∞-values correspond well to the simulation results in Figure 3, which suggests that they are useful for predicting the behaviors of the service time distributions.

3.3 Different actuators

The only actuator considered so far in our modeling has been the server speed u. However, our model structure can easily be modified to support other actuators as well. One example is admission control (see e.g. [Kihl et al., 2004]), where the arrival rate λ is affected by the choice of admitting or denying the incoming server requests. In this case, with server speed $u = 1$, our model structure (8) would change to:

$$\begin{aligned} \dot{w} &= u_\lambda \, \bar{x} - 1 \\ \dot{q} &= u_\lambda - k \, \frac{q}{w}, \end{aligned} \tag{13}$$

with $u_\lambda \geq 0$ the admitted arrival rate.

Another example of a different actuator in the field of cloud computing is graceful degradation, where the quality of the served application content is sacrificed when necessary in order to maintain predictable response times. One realization is the Brownout concept [Klein et al., 2014; Nylander et al., 2018], where the application is split into two parts, one that is always computed (mandatory), and one which is computed if possible (optional). The choice of serving optional content is thus the control signal, and affects the mean service time $\bar{x}$ of the application. Here, the model structure (8) thus changes to

$$\begin{aligned} \dot{w} &= \lambda \, u_{\bar{x}} - 1 \\ \dot{q} &= \lambda - k \, \frac{q}{w}, \end{aligned} \tag{14}$$

with $u_{\bar{x}} \geq 0$ the mean service time under graceful degradation.

3.4 Generalizing to the LPS discipline

Our modeling approach so far has been focused on describing the queue length behavior for the queuing discipline PS. However, one important property of our model structure (8), is that it reduces to a pure integrator model for $k = 1$. This allows our structure to perfectly describe the exponential distribution under PS, but also any distribution under FCFS. Thus, for the more general queuing discipline LPS, the value of k can be chosen to represent any M_C number of simultaneous requests. Values of k closer to 1 suggests a behavior close to FCFS (with M_C close to 1), and k further away from 1 suggests a behavior closer to PS. When using our model structure for a real

cloud server setup, the estimation of k can be performed regardless of the true behavior of the server. This is an advantage as most servers do not follow the PS or FCFS model exactly, they are most likely somewhere in between.

3.5 Estimations

Extensive work on estimator design for the involved components of our model (8) is out of scope for this paper. However, we do want to mention what states and constants that need to be estimated, in order to be able to design a controller: (i) The arrival rate λ; (ii) Mean service time $\bar{x}$; (iii) Remaining work w; and (iv) The distribution parameter k.

The estimation of λ is straightforward as it only implies measuring the arrivals of the requests, whereas the other states and constants require slightly more effort. In order to estimate the remaining work w for the general LPS discipline, the concurrency value M_C also has to be estimated. This can be done in a fashion similar to the approach used in [Keith et al., 2019].

4. Model Evaluation

In this section, we evaluate our proposed dynamic queue model and compare it to two important existing models. We perform the evaluation in our simulator with Poisson arrivals using PS as the queuing discipline in Section 4.2, and then extend to the generalized LPS discipline in Section 4.3.

4.1 The simulator

The simulator used in this paper is based on the Brownout simulator[2], first used in the paper by Dürango et. al. [Dürango et al., 2014]. The simulator allows us to, for each scenario, define (i) inter-arrival time distribution F_{arr}; (ii) service time distribution F; and (iii) queuing discipline (FCFS, PS or LPS). The requests are processed in a discrete-event based fashion. For a more complete description of the simulator, we refer to [Dürango et al., 2014].

4.2 Comparison to existing models

In our evaluation, we compare our proposed model (8), denoted as $\mathcal{M}_N$, to the following two models: (i) The integrator model $\mathcal{M}_I$ based on (6); and (ii) The Paganini model $\mathcal{M}_P$ from Equation (7). However, as the integrator model (6) lacks parameterization, we introduce a gain parameter k_I, and modify (6) to the following:

$$\mathcal{M}_I : \qquad \dot{q} = \lambda - \frac{k_I}{\bar{x}}\, u. \tag{15}$$

[2] https://github.com/cloud-control/brownout-simulator

As the initial simulation study showed that, for the PS discipline, the queue length dynamics depends heavily on the chosen service time distribution, we perform the evaluation using five different distributions. First, two that result in $k > 1$:

- Weibull $F_{\text{wei}}(x) = 1 - e^{-(2.0\,x/\bar{x})^{0.5}}, \quad x \geq 0$
- Hyperexponential $F_{\text{hyp}}(x) = \sum_{i=1}^{2} p_i \left(1 - e^{-x/\bar{x}}\right)$, with $x \geq 0$, $p_1 = 0.87$ and $p_2 = 1 - p1 = 0.13$.

Second, two that result in $k < 1$:

- Uniform $F_{\text{uni}}(x) = \begin{cases} x/2\bar{x} & \text{if } 0 \leq x \leq 2\bar{x} \\ 1 & \text{if } x > 2\bar{x} \end{cases}$
- Deterministic $F_{\text{det}}(x) = \begin{cases} 0 & \text{if } 0 \leq x < \bar{x} \\ 1 & \text{if } x \geq \bar{x} \end{cases}$

Finally, one that results in $k = 1$:

- Exponential $F_{\text{exp}}(x) = 1 - e^{-x/\bar{x}}, \quad x \geq 0.$

To ensure that we get a general evaluation of the models, we perform 1000 randomized scenarios using the following parameters:

- Mean service time $0.01 \leq \bar{x} \leq 0.1$
- Arrival rate $\lambda = 0.5\,\frac{1}{\bar{x}}$,

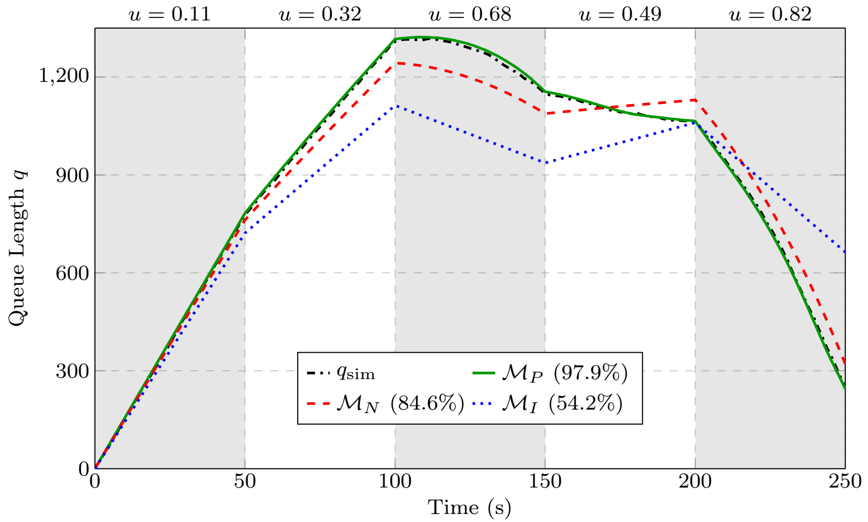
such that for server speed $u = 0.5$, the queues are stationary.

Each scenario consists of an open loop sequence of five randomly chosen server speeds $\{u_1, u_2, u_3, u_4, u_5\}$, where each speed is maintained for 50 seconds, resulting in a total scenario length of 250s. The intervals for the server speeds are chosen to ensure that we do not get empty servers.

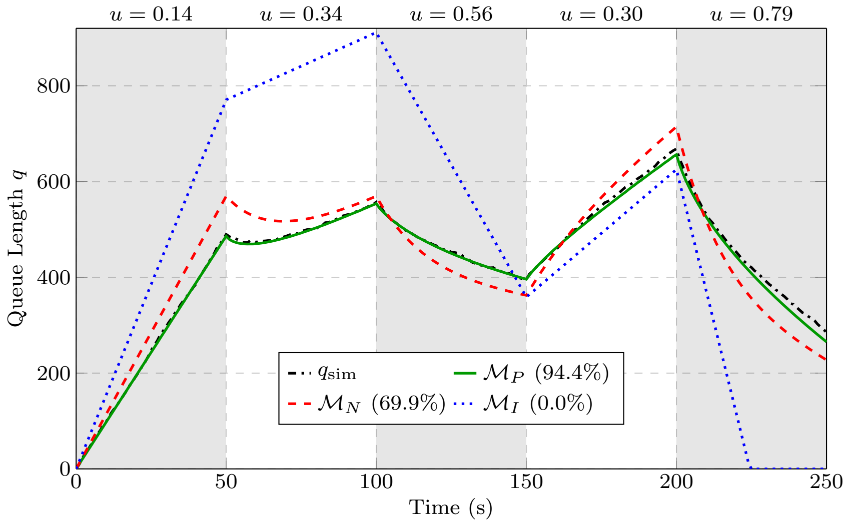
In each scenario, we: (i) Run 20 repeated simulations with different random seeds; (ii) Form the average queue length behavior q_{sim}; (iii) Numerically solve the Paganini model; and (iv) Calculate the best fit values for $\mathcal{M}_I$ and $\mathcal{M}_N$, assuming static k-values during the scenario. The model fit percentages $0\% \leq m_f \leq 100\%$ are calculated based on the normalized root mean squared error criterion (NRMSE):

$$m_f = 100 \cdot \left(1 - \frac{||\, q_{\text{sim}} - q \,||}{||\, q_{\text{sim}} - \mathbf{E}\left(q_{\text{sim}}\right) ||}\right) \%. \tag{16}$$

Note that a fit $m_f = 0\%$ represents a model that performs exactly as poor as a straight line with value $\mathbf{E}(q_{\text{sim}})$. Fit percentages below zero are also



(a) Uniform distribution F_{uni}.



(b) Weibull distribution F_{wei}.

Figure 4. Two example scenarios under the PS queuing discipline.

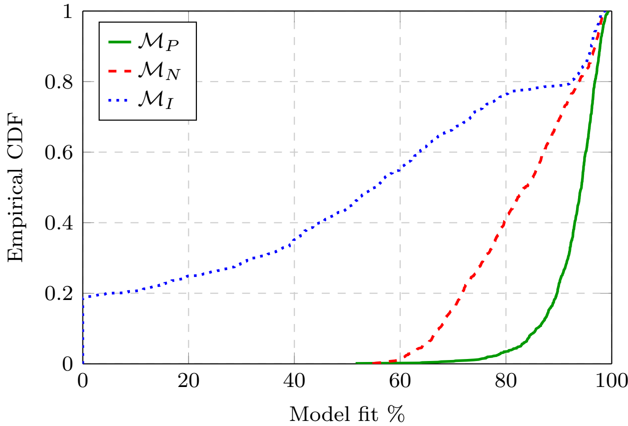


Figure 5. Statistical results of the model fits for PS.

possible, but for simplicity and readability we have decided to saturate them to zero.

Two example scenarios out of the 1000 evaluated are shown in Figure 4, where Figure 4(a) was performed under the uniform distribution F_{uni} and 4(b) under Weibull F_{wei}. The results show queue length plotted against time for the simulated average q_{sim}, together with the best model fits for the evaluated models $\mathcal{M}_N$, $\mathcal{M}_I$ and $\mathcal{M}_P$. As expected, since the Paganini model is theoretically exact for the PS discipline, it outperforms the other two here. Our proposed model $\mathcal{M}_N$ performs reasonably well for both scenarios, whereas the integrator model $\mathcal{M}_I$ is not able to describe the dynamics in the Weibull scenario at all, resulting in a 0% fit.

The complete statistical results of the model fits of all 1000 scenarios are available in Figure 5. It shows the empirical cumulative distribution function (CDF) of the model fits for the three evaluated models. As a fit close to 100% is desirable, the further to the right the line is plotted the better. The results show that, for all scenarios, the Paganini model $\mathcal{M}_P$ performs very well (fits 80-100%), and our proposed model $\mathcal{M}_N$ reasonably well with a slightly larger spread (fits 60-100%). The integrator model $\mathcal{M}_I$, however, exhibits a very wide spread of fits (0-100%), where about a fifth of all scenarios resulted in a zero fit percentage.

The model fits of the 1000 scenarios are broken down into average values per distribution in Table 3. The most interesting results are for the integrator model $\mathcal{M}_I$, where it can be seen that it only describes the dynamics well for the exponential distribution, and reasonably well for uniform. The other distributions range from bad to worse, and for Weibull the average is as low as 0.0%. For the Paganini model $\mathcal{M}_P$, the fits are high for all distributions, but

Table 3. Statistical results for PS model fits per distribution.

	Average Model Fit %				
	F_{wei}	F_{hyp}	F_{uni}	F_{det}	F_{exp}
$\mathcal{M}_P$	89.5	90.6	94.3	95.3	92.7
$\mathcal{M}_N$	73.9	83.8	88.2	70.3	96.3
$\mathcal{M}_I$	0.0	23.5	69.0	50.6	95.6

lowers slightly for hyperexponential and Weibull, that exhibit more stochastic variations. Our proposed model $\mathcal{M}_N$ performs very well for exponential, and reasonably well for the other distributions.

Recall that all model fits presented thus far have been calculated for static k-values for both models $\mathcal{M}_I$ and $\mathcal{M}_N$ ($\mathcal{M}_P$ has no parametrization). In other words, if an estimation scheme would try to update the k-values online, the model fits would become higher. For our proposed model $\mathcal{M}_N$ the results for the static k-values suggest that the estimations do not have to be exact in order for the model structure to represent the dynamics fairly well (fits 60-100%).

4.3 Generalized queuing discipline

In this section we repeat the exact same evaluation procedure from Section 4.2, but using the generalized queuing discipline LPS instead. We simulate two cases of LPS, dynamically setting M_C as a ratio r_M^C of the current queue length q. We use ratios $r_M^C = 0.33$ and $r_M^C = 0.67$. Normally, the M_C value of a real-world server is assumed to be constant. Furthermore, for a closed loop scenario, the server queue length would be kept close to the set-point at all times, resulting in an approximately constant ratio between the M_C value and queue length. However, as we test open loop scenarios with large variations in queue lengths, keeping a constant M_C in our tests would result in large variations in the ratio r_M^C. To emulate a real-world closed loop setting we thus dynamically change the M_C value in our simulations, in order to keep a constant ratio r_M^C for our two LPS cases. For comparison purposes, we also simulate a true FCFS case with a constant $M_C = 1$.

The statistical results of the LPS evaluation are available in Figure 6. Just as in Figure 5, each plot 6(d)-6(a) shows the empirical CDF of the model fit percentages for all 1000 scenarios for the Paganini model $\mathcal{M}_P$, our proposed model $\mathcal{M}_N$ and the integrator model $\mathcal{M}_I$. The results show that the Paganini model is only able to describe the true PS case well, for all other cases it exhibits very poor results. This is not surprising as it is only designed for true PS, and with no parameterization, its behavior will only fit well with PS queues. The integrator model, on the other hand, shows very high fits for the FCFS case, and then gets much worse as the cases get closer

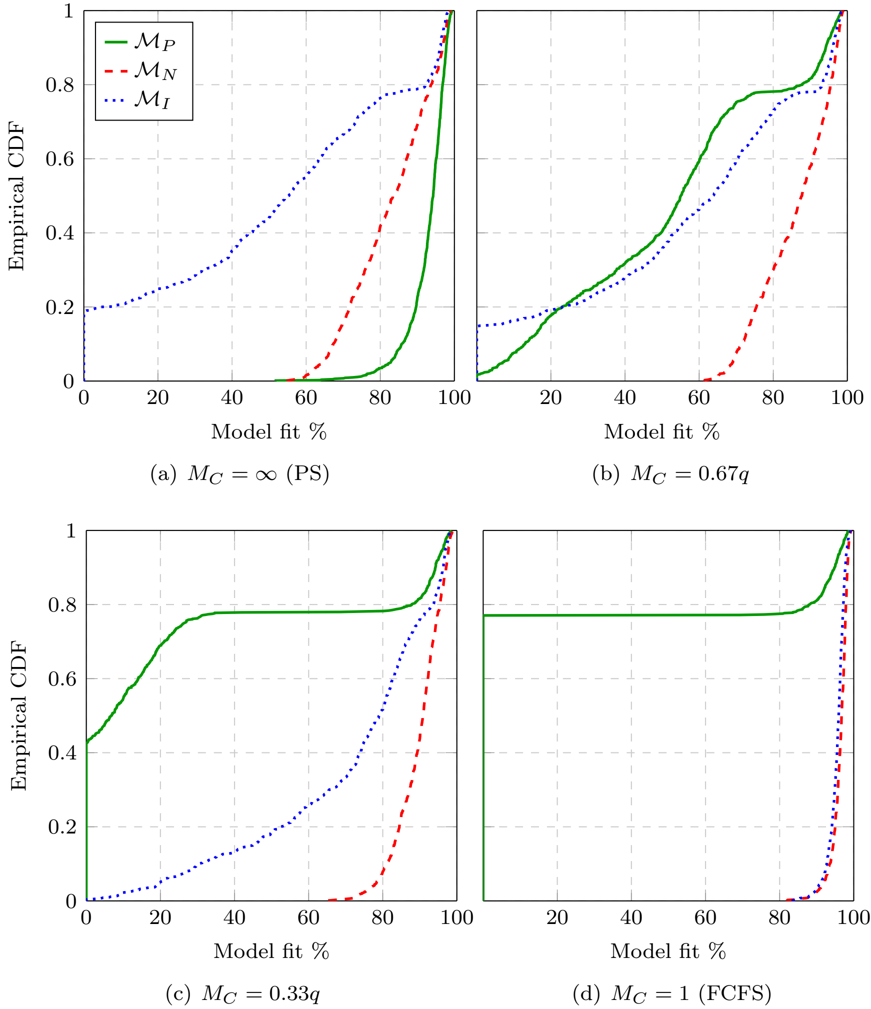


Figure 6. Statistical results comparison for model fits between different values for M_C. The legend applies to all plots.

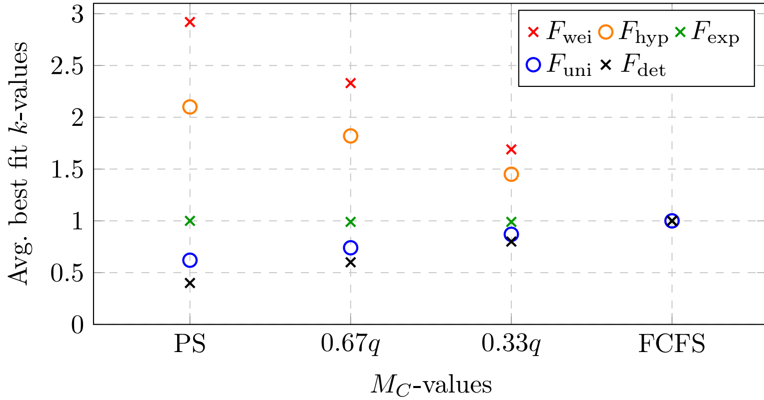


Figure 7. Best-fit k-values for $\mathcal{M}_N$ under different M_C values.

to PS. These results align well with the fact that FCFS queues behave like integrators, see e.g. Equation 2.

Our proposed model $\mathcal{M}_N$ is the only one that exhibits acceptable fits for all four cases. The fits are slightly higher for the cases closer to FCFS, due to their more simplistic, integrator-like, behavior, but also the PS-like cases perform well. This adaptability of our proposed model structure is due to the fact that the k-value can be set to describe the dynamics for both FCFS, PS, and everything in between. The structure is originally designed for true PS, but for cases closer to FCFS, the best fit k-value simply tends closer to 1. This is illustrated in Figure 7, where the average best fit k-values are plotted for the five distributions under different concurrency values M_C.

5. Control Design

The purpose of this section is to provide a queue length control design example that highlights the insights that can be gained from our proposed model $\mathcal{M}_N$. We choose to base the design on the linearization of our model, with transfer function $G_N(s)$ re-stated for convenience:

$$G_N(s) = -\frac{q_0}{w_0} \frac{\left(1 + s\frac{w_0}{u_0}\right)}{s\left(1 + s\frac{w_0}{ku_0}\right)} = K_P \frac{(1 + sT_z)}{s\,(1 + sT_p)}, \tag{17}$$

i.e. with process parameters $K_P = -\frac{q_0}{w_0}$, $T_z = \frac{w_0}{u_0}$ and $T_p = \frac{w_0}{ku_0}$. As can be seen in (17), the distribution specific k-parameter decides the location of the process pole $p_1 = \frac{-1}{T_p} = \frac{-ku_0}{w_0}$, in relation to the zero $z_1 = \frac{-1}{T_z} = \frac{-u_0}{w_0}$.

For distributions with $k < 1$, $|p_1| < |z_1|$, and vice versa. This is visualized in the Bode diagram shown in Figure 8. The diagram shows a linearization performed with $q_0 = 100$, $w_0 = 2$ and $u_0 = 0.5$. The figure shows the magnitude and phase of $G_N(s)$ for two example distributions, with k-values 0.5 (F_{uni}, blue) and 3.5 (F_{wei}, red). For $k = 0.5$, the pole p_1^{uni} appears to the left of zero z_1^{uni} (i.e. p_1^{uni} has lower frequency) in the Bode diagram, thus resulting in a phase drop of 20 degrees. For $k = 3.5$, we instead see that the zero z_1^{wei} appears to the left of p_1^{wei}, leading to a phase increase of about 30 degrees. The differences in phase between the two distributions are greatest between p_1 and z_1, and we choose to denote this frequency range as the critical range $\{w_{\text{critical}}\}$. In Figure 8, $\{w_{\text{critical}}\}$ is shown for the Weibull distribution, however, the example is constructed such that both distributions have the approximately same critical frequency range. The closer to $\{w_{\text{critical}}\}$ the cut-off frequency of the control design is chosen, the more the characteristic of the distribution will affect the design.

As distributions with $k > 1$ result in a phase increase at the critical frequency range, it can be concluded that distributions with $k > 1$ are in fact easier to control, compared to their $k < 1$ counterparts! As no phase compensation is needed for $k > 1$, simpler control structures can thus be allowed. This is an interesting insight that will be further examined in the following loop shaping design example.

In our example design, we consider a PID control structure with a first order low pass filter on the derivative part:

$$C(s) = K\left(1 + \frac{1}{sT_i} + \frac{sT_d}{1 + sT_d/N}\right), \tag{18}$$

with proportional gain K, integral time constant T_i, derivative time constant T_d and filter constant N. As our model structure (8) is non-linear, we use an adaptive approach where the system is linearized at each time step. The controller parameters are designed with respect to the linearized model $G_N(s)$, using a loop shaping based approach. The goal of the example design is not to be optimal, but to provide a way to compare the design needs for different service time distributions, i.e. with k less than or greater than 1.

As a first step, consider a PI design that sets the integral time constant T_i such that the phase at the desired cut-off frequency ω_c is lowered by $p_i > 0$ radians:

$$T_i = \frac{\tan\left(-p_i + \pi/2\right)}{\omega_c}. \tag{19}$$

The proportional gain K is then chosen with negative sign as $K_P < 0$, and with magnitude to obtain the desired ω_c:

$$|K| = \frac{\sqrt{1 + \omega_c^2 T_p^2}}{|K_P||\hat{C}(i\omega)|\sqrt{1 + \omega_c^2 T_z^2}}, \tag{20}$$

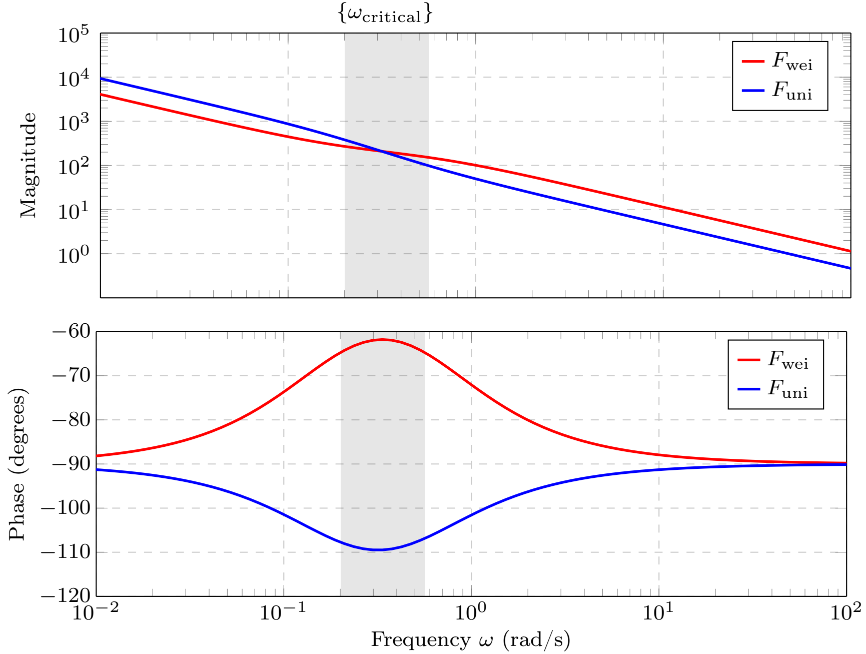


Figure 8. Example Bode diagram of $G_N(s)$.

where $\hat{C}$ is the controller except for its proportional gain K. This method thus results in a loop shaping design that lowers the phase margin by p radians, regardless of the initial phase margin in the open loop system. This allows us to compare how distributions with different k-values can tolerate phase margin drops due to increased integral action.

The final design of our controller includes the derivative part, composed of its derivative time constant T_d and its filter constant N. These two parameters are designed such that the phase at ω_c is lifted by p_d radians, and such that the derivative gain at higher frequencies is attenuated. We choose N as the parameter that sets the filter pole at a frequency k_f times higher than ω_c:

$$N = k_f \omega_c T_d. \tag{21}$$

Solving for a derivative part that lifts the phase by p_d radians does not result in an explicit expression for T_d, thus it is not shown here. Instead, we solve it numerically in our simulator during run time. We denote these PI and PID designs as $C_{\mathcal{M}_N}^{PI}$ and $C_{\mathcal{M}_N}^{PID}$.

In order to realize the implementations of the controllers, the involved process parameters K_P, T_z and T_p need to be estimated. These are in turn

composed by the states q_0, u_0, w_0 and the distribution specific parameter k. As both queue lengths q and control signals u are measurable, only w and k need to be estimated. For w, we use a method that measures arrival and departure times of all requests to estimate a CDF of the service time distribution. From the CDF it is then straightforward to estimate the remaining work for each request, and then to form an estimate of w as the sum of the estimated remaining work of each request. For k, we use our non-linear model structure 8, with k as the only unknown, to form a simple exponentially weighted moving average filter that provides an estimate of k.

6. Control Evaluation

To simplify the evaluation of our example design, we will focus entirely on the PS queuing discipline in this section. However, the same insights also apply to the more general LPS discipline. The evaluation will be performed using the same discrete-event simulator as in Section 4.

The example adaptive loop shaping based designs $C^{PI}_{\mathcal{M}_N}$ and $C^{PID}_{\mathcal{M}_N}$, including both the PI and PID structure, from Section 5 will be evaluated and compared to a corresponding PI/PID control design , $C^{PI}_{\mathcal{M}_I}$ and $C^{PID}_{\mathcal{M}_I}$ respectively, based on the integrator model $\mathcal{M}_I$ from Equation (15).

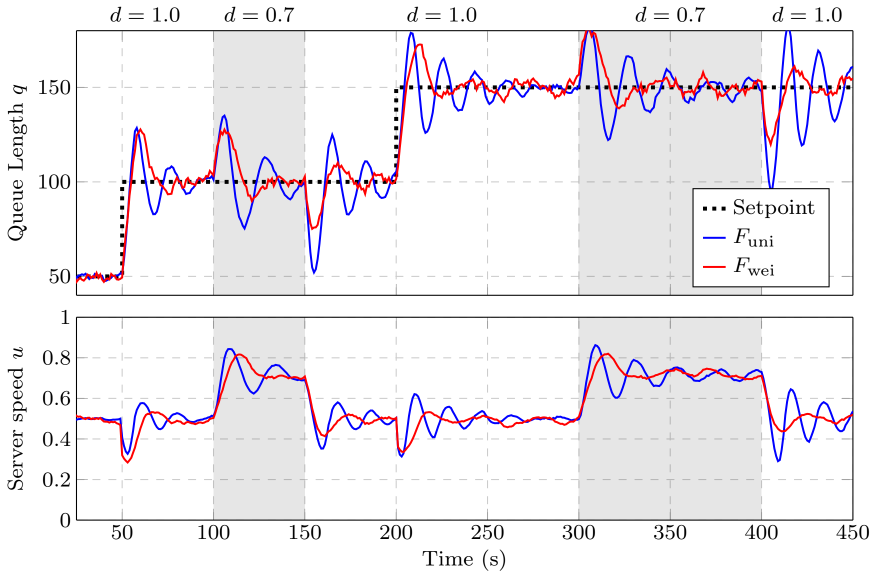
It uses the same expressions and calculations for T_i as in (19) and T_d, but determines K from the simple integrator model $\mathcal{M}_I$ instead as

$$|K| = \frac{1}{|k_I||\hat{C}(i\omega)|}, \tag{22}$$

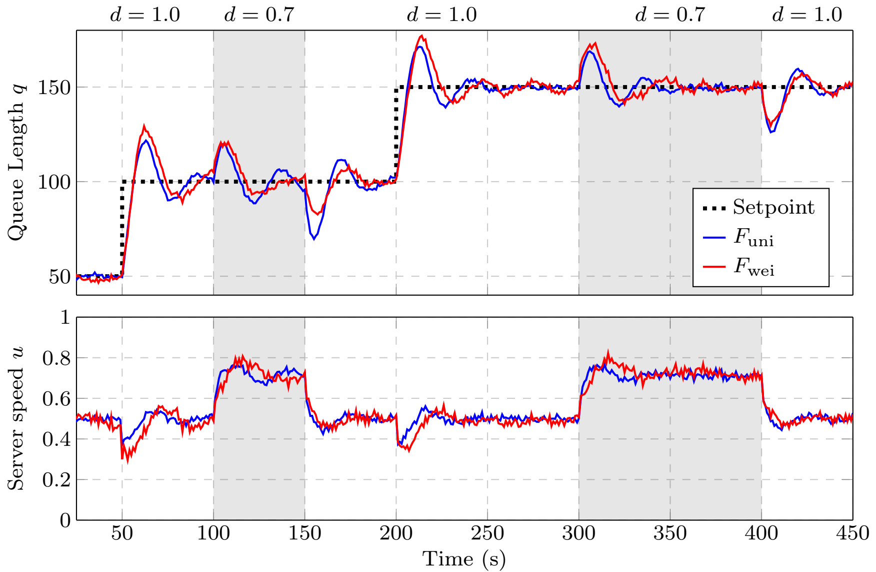
in order to obtain the desired ω_c.

We evaluate the four control designs for two different service time distributions, Weibull (F_{wei}, $k > 1$) and Uniform (F_{uni}, $k < 1$). In the first simulation experiment, we use a cut-off frequency $\omega_c = 0.3$, that results in a cut-off frequency that is right in the center of the critical frequency range $\{w_{\text{critical}}\}$. The integral time constant T_i is designed to obtain a phase drop of $p_d = \pi/3$ radians, and the derivative time constant T_d to lift the phase by the same amount. The filter constant N is set to $3\omega_c T_d$, i.e. with $k_f = 3$. We use a sequence with queue length setpoints $\{50, 100, 150\}$ and load changes $d = \{1.0, 0.7, 1.0, 0.7, 1.0\}$. The value 0.7 denotes a server slowdown of 30%, and thus acts as a load disturbance from 1.0 that represents no slowdown.

The results of the first experiment are shown in Figures 9 and 10. Figure 9 shows the results for the control designs based on our proposed $\mathcal{M}_N$ model. The $C^{PI}_{\mathcal{M}_N}$ design is shown in the left subfigure, whereas $C^{PID}_{\mathcal{M}_N}$ is shown in the right. The upper plots show queue lengths q and the lower show server speeds u, i.e. the control signals. For the PI designs, a significant difference can be noted between the two distributions F_{wei} and F_{uni}. For the Uniform

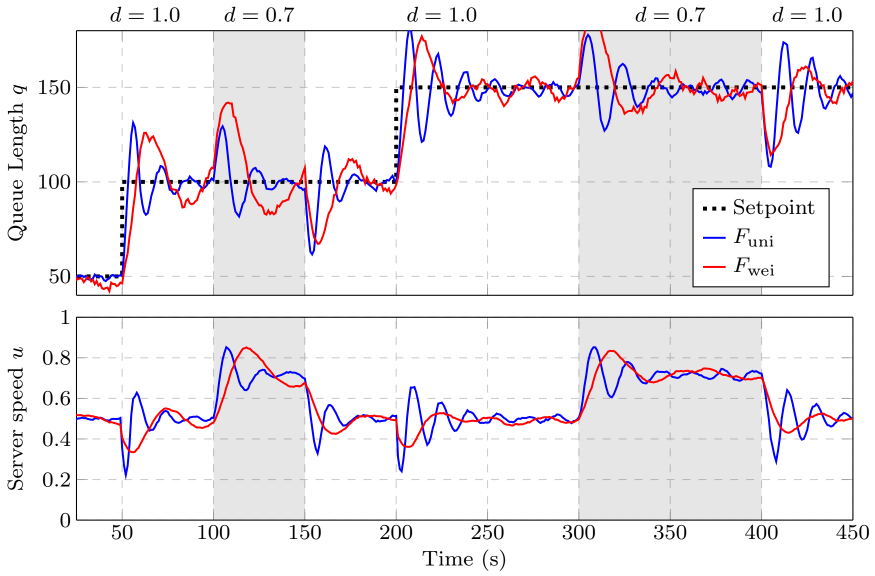


(a) PI control with $\omega_c = 0.3$. Legend applies to both plots.

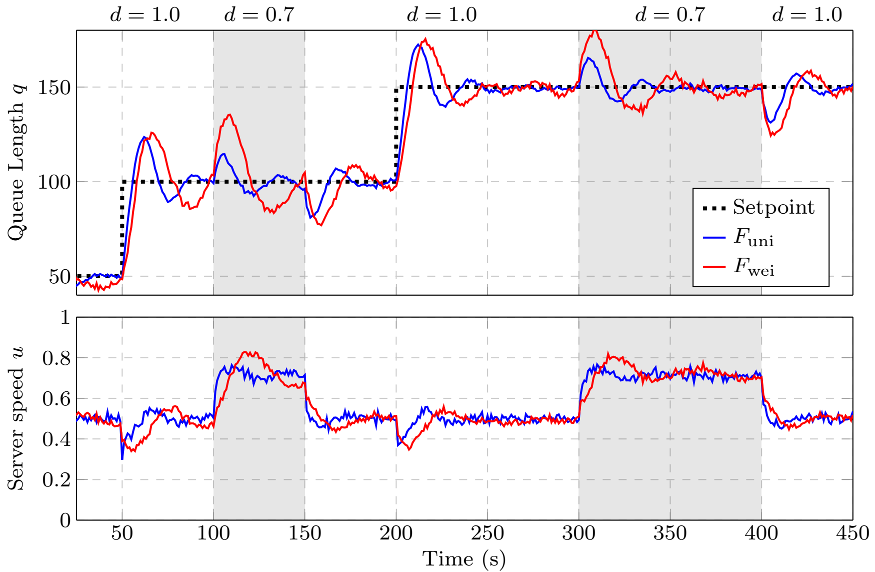


(b) PID control with $\omega_c = 0.3$. Legend applies to both plots.

Figure 9. Control designs based on the $\mathcal{M}_N$ model.



(a) PI control with $\omega_c = 0.3$. Legend applies to both plots.



(b) PID control with $\omega_c = 0.3$. Legend applies to both plots.

Figure 10. Control designs based on the $\mathcal{M}_I$ model.

distributions, large oscillations occur both at setpoint changes and at load disturbances, whereas for Weibull only relatively small over and undershoots can be seen. It is thus clear that in this design example, the Uniform distribution can not tolerate integral action well, as it lowers the phase margin.

Looking at the $\mathcal{M}_N$ PID designs, the differences between the two distributions become much smaller. Here, Uniform and Weibull behave almost identically, and it is clear that the introduction of a derivative part increases the performance for Uniform greatly. At the same time, the results for Weibull are barely affected by the derivative action. This is of course due to the fact that the Weibull distribution has a much larger phase margin to start with. The example Bode diagram in Figure 8 in fact shows a linearization performed at approximately 80 seconds into all experiments run in Figures 9-10. It can thus be seen that the cut-off frequency $\omega_c = 0.3$ is located in $\{w_{\text{critical}}\}$, and the differences in phase between the distributions are maximized. The phase lifting effect of the derivative part thus only makes a difference for the Uniform distribution, where Figure 8 shows that it has a phase drop in the critical range. It should of course be noted that these highlighted differences between the two distributions in this example are large due to the chosen cut-off frequency ω_c.

Compared to the results in Figure 9, a very similar behavior is also observed in the designs based on the $\mathcal{M}_I$ model, shown in Figure 10. This is expected, as these designs only differ from the $\mathcal{M}_N$ based designs by the determination of the proportional constant K. The main difference is, however, that the differences in behavior between the two distributions Weibull and Uniform can not be explained by the simpler integrator model $\mathcal{M}_I$. Our proposed model structure $\mathcal{M}_N$ can, through its linearization (17), provide much clearer insights on where simpler or more complicated control structures are needed. If a cut-off frequency ω_c is desired close to the pole and zero of the distributions, the differences of the distributions will be highlighted and the controller designs will have to take that into account. However, using our linearized model (17), it is also possible to identify frequency ranges where simpler controllers will work for all distributions. In these ranges, the model can be safely represented as a simple integrator.

This is demonstrated in Figure 11, where both slow ($\omega_c = 0.05$) and fast ($\omega_c = 10$) PI controllers based on our proposed $\mathcal{M}_N$ model are shown. Both of these cut-off frequencies are chosen to be far away from the critical frequency range $\{w_{\text{critical}}\}$. The scenarios performed here are identical to the experiments presented in Figures 9 - 10, except for the load disturbance sequence d in the slower design that only uses a 10% slowdown due to its poorer performance. Furthermore, the control designs are identical to the previous designs with the exception of the determination of K, which is set to obtain the new desired cut-off frequencies ω_c.

The faster control with $\omega_c = 10$ is shown in Figure 11(b), and at this

speed it is clear that the differences between the two distributions is almost completely gone. The Uniform distribution still has slightly larger oscillations during the setpoint changes, but other than that the behaviours are very similar. Figure 11(a) shows the slower design with $\omega_c = 0.05$, and here the behaviours are almost identical during the setpoint changes, but not during the load changes. However, the Uniform distribution is not particulary more oscillatory than Weibull, but the control design seems to be less efficient for the load disturbances as its deviations from the queue length setpoint are larger. This can be explained by our chosen loop shaping control design, which focuses on keeping the same cut-off frequency ω_c for both distributions. However, this does not guarantee that all transfer functions become identical. As the impact of load disturbances is determined in this case by the transfer function $\frac{G_N(s)}{1+G_N(s)C(s)}$, it is natural that the two distributions will behave differently.

Despite the differences that can be observed in the slow and fast scenarios in Figure 11, it is clear that both distributions are well controlled by simple PI controllers as can be anticipated by studying the linearization (17) of our model structure.

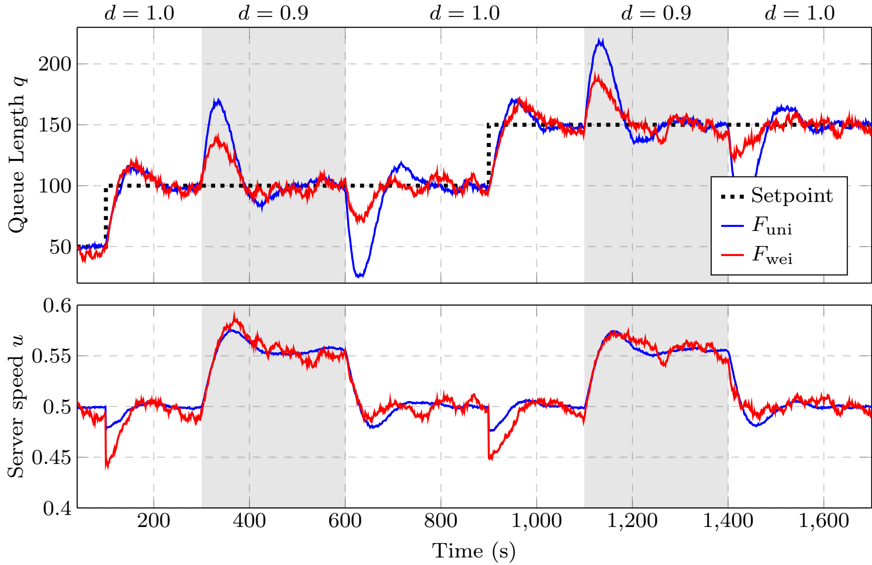
7. Conclusion

In this paper, we have presented an overview of server queue models, focusing on dynamic models for controlled servers queues subject to actuators. In the initial simulation study, we have shown that a simple integrator model is not sufficient for processor sharing queues, where the dynamics depend heavily on the distribution of service times.

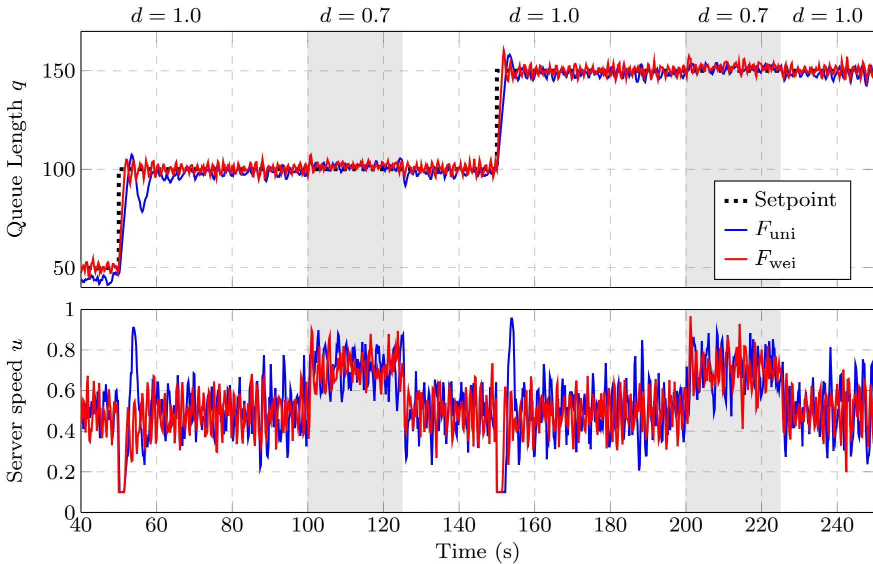
We have proposed a novel model structure designed for processor sharing queues, that can also be adapted for the more general LPS discipline under any concurrency value M_C, through the structure's k-value. This allows our model to be useful for describing the dynamics of real-world servers that not necessarily fit the basic queuing disciplines FCFS and PS. Additionally, the model structure is nonlinear, yet simple enough to be useful for control design and online estimations.

Through an extensive simulation campaign with a randomized scenario approach, we have shown empirically that our proposed model structure is useful for a wide range of service time distributions. Furthermore, it outperforms both the integrator model and the Paganini model for the LPS cases where the concurrency value is in between the two extremes FCFS ($M_C = 1$) and PS ($M_C = \infty$).

Finally, using a loop shaping control design example we have shown how our model structure can, through its linearization, be used to gain insight into what frequency range the service time distributions impact the behav-



(a) PI control with $\omega_c = 0.05$. Legend applies to both plots.



(b) PI control with $\omega_c = 10$. Legend applies to both plots.

Figure 11. Slower and faster control designs based on the $\mathcal{M}_N$ model. Notice the differences in scale.

ior considerably. In and close to this critical frequency range $\{w_{\text{critical}}\}$, the controllers need to be designed with respect to the characteristics of the distribution. However, outside this critical range, our model structure also shows that simpler models, such as the integrator model, can be used to design well performing PI controllers with good robustness margins only considering the mean values of the service time distributions.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Nordforsk Nordic Hub on Industrial IoT (HI2OT), and by the ELLIIT Excellence Center at Lund University.

References

Abdelzaher, T. F. and Chenyang Lu (2000). "Modeling and performance control of internet servers". In: *Proceedings of the 39th IEEE Conference on Decision and Control.* Vol. 3, pp. 2234–2239. DOI: 10.1109/CDC.2000.914129.

Abdelzaher, T. F., J. A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu (2003). "Feedback performance control in software services". *IEEE Control Systems Magazine* **23**:3, pp. 74–90. DOI: 10.1109/MCS.2003.1200252.

Arcelli, D., V. Cortellessa, A. Filieri, and A. Leva (2015). "Control theory for model-based performance-driven software adaptation". In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures - QoSA '15.* ACM Press.

Barroso, L. A. and U. Hölzle (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool.

Dürango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with Brownout". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014.* CDC14, pp. 5320–5327.

Jean-Marie, A. and P. Robert (1994). "On the transient behavior of the processor sharing queue". *Queueing Systems* **17**:1-2, pp. 129–136.

Keith, A., D. Ahner, and R. Hill (2019). "An order-based method for robust queue inference with stochastic arrival and departure times". *Computers & Industrial Engineering* **128**, pp. 711–726. DOI: `10.1016/j.cie.2019.01.005`. URL: `https://doi.org/10.1016/j.cie.2019.01.005`.

Kihl, M., A. Robertsson, and B. Wittenmark (2004). "Control theoretic modelling and design of admission control mechanisms for server systems". In: Mitrou, N. et al. (Eds.). *Networking 2004*.

Kim, D. H., C. Imes, and H. Hoffmann (2015). "Racing and pacing to idle: theoretical and empirical analysis of energy optimization heuristics". In: *IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*. IEEE. DOI: `10.1109/cpsna.2015.23`. URL: `https://doi.org/10.1109/cpsna.2015.23`.

Kitaev, M. Y. and V. V. Rykov (1995). *Controlled queueing systems*. CRC Press.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: Building more robust cloud applications". In: *36th International Conference on Software Engineering*. ICSE14. ACM, Hyderabad, India, pp. 700–711. ISBN: 978-1-4503-2756-5.

Kleinrock, L. (1975). *Queueing Systems. Vol. I: Theory*. Wiley Interscience.

Kulkarni, V. G. (1997). "Fluid models for single buffer systems". *Frontiers in queueing: Models and applications in science and engineering* **321**, p. 338.

Nylander, T., C. Klein, K.-E. Årzén, and M. Maggio (2018). "Brownout^{CC}: Cascaded control for bounding the response times of cloud applications". In: *2018 American Control Conference*. Milwaukee, Wisconsin, USA.

Paganini, F., A. Tang, A. Ferragut, and L. L. H. Andrew (2012). "Network stability under alpha fair bandwidth allocation with general file size distribution". *IEEE Transactions on Automatic Control* **57**:3, pp. 579–591.

Patikirikorala, T., A. Colman, J. Han, and L. Wang (2012). "A systematic survey on the design of self-adaptive software systems using control engineering approaches". In: SEAMS '12. IEEE Press, Zurich, Switzerland, pp. 33–42. ISBN: 9781467317870.

Perez, J. F. and G. Casale (2017). "Line: evaluating software applications in unreliable environments". *IEEE Transactions on Reliability* **66**:3, pp. 837–853. DOI: `10.1109/tr.2017.2655505`. URL: `https://doi.org/10.1109/tr.2017.2655505`.

Pothukuchi, R. P., S. Y. Pothukuchi, P. G. Voulgaris, and J. Torrellas (2020). "Control systems for computing systems: making computers efficient with modular, coordinated, and robust control". *IEEE Control Systems Magazine* **40**:2, pp. 30–55. DOI: `10.1109/MCS.2019.2961733`.

Schwarz, J. A., G. Selinka, and R. Stolletz (2016). "Performance analysis of time-dependent queueing systems: survey and classification". *Omega* **63**, pp. 170–189. DOI: 10.1016/j.omega.2015.10.013. URL: https://doi.org/10.1016/j.omega.2015.10.013.

Wang, W.-P., D. Tipper, and S. Banerjee (1996). "A simple approximation for modeling nonstationary queues". In: *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*. IEEE Comput. Soc. Press.

Zhang, J., J. Dai, and B. Zwart (2009). "Law of large number limits of limited processor-sharing queues". *Math. Oper. Res.* **34**, pp. 937–970.

Paper III

Cloud Application Predictability through Integrated Load-Balancing and Service Time Control

Tommi Nylander **Marcus Thelander Andrén**
Karl-Erik Årzén **Martina Maggio**

Abstract

Cloud computing provides the illusion of infinite capacity to application developers. However, data center provisioning is complex and it is still necessary to handle the risk of capacity shortages. To handle capacity shortages, graceful degradation techniques sacrifice user experience for predictability. In all these cases, the decision making policy that determines the degradation interferes with other decisions happening at the infrastructure level, like load-balancing choices. Here, we reconcile the two approaches, developing a load-balancing strategy that also handles capacity shortages and graceful degradation when necessary. The proposal is based on a sound control-theoretical approach. The design of the approach avoids the pitfalls of interfering control decisions. We describe the technique and provide evidence that it allows us to achieve higher performance in terms of emergency management and user experience.

© 2018 IEEE. Originally published in IEEE International Conference on Autonomic Computing (ICAC), Trento, Italy, September 2018. Reprinted with permission. The article has been reformatted to fit the current layout.

1. Introduction

Capacity provisioning is of crucial importance in modern distributed computation infrastructures. To determine the size of data centers, and properly dimension the resources to be allocated in each geographic location, most data center owners use predictions of the computational needs [Lin et al., 2018; Östberg et al., 2017]. The computational resource within a data center is then used to serve requests coming from multiple clients, providing the illusion of infinite capacity and, as a result, the possibility of bounding the latency [Wang et al., 2016; Björkqvist et al., 2018; Björkqvist et al., 2016; Javadi and Gandhi, 2017; Ghahremani et al., 2017; Kaler et al., 2017]. To do so the architecture uses multiple instances of the same application, here called *replicas*, and predictions and estimations of traffic and needed computational capacity.

The predictions of the incoming traffic and the corresponding estimates [Grimes et al., 2016; Grohmann et al., 2017] of the required computational capacity are necessarily subject to errors and uncertainty [Bencomo and Belaggoun, 2014]. The presence of these errors naturally leads to two possible management strategies. The first strategy is over provisioning [Greenberg et al., 2008; Xue et al., 2016]. Over provisioning increases the management cost for a cloud application, but guarantees user satisfaction. The second strategy is provisioning according to expectations and handling capacity shortages via user experience degradation [Neumann, 2009; Ding et al., 2011; Tomás and Tordsson, 2014; Breitgand and Epstein, 2012; Klein et al., 2014a], or via approximate computing [Perez et al., 2017; Sun et al., 2017; Hoger and Kao, 2016]. Generally speaking, these ways of handling capacity shortages are typically clustered under the umbrella of *graceful degradation*.

Graceful degradation techniques involve taking corrective actions (that typically degrade the user experience) to ensure that the computing platform achieves predictability (for example, that any request receives a response within a given time). For example, Brownout [Klein et al., 2014a] sacrifices the quality of the response given to users to ensure that a large fraction of the requests experience a predictable latency. Brownout is based on a control approach [Litoiu et al., 2013; Filieri et al., 2017; Maggio et al., 2017], and a controller selects – at the replica level – requests to be answered with full quality (both the mandatory and the optional part of the response are computed) and requests to be given an approximate answer (only the mandatory part is computed). The approach has proven to be successful to bound the response times of single machines. It was then combined with load-balancing strategies [Dürango et al., 2014; Klein et al., 2014b], showing that the control strategy at the replica level and the load balancer could interfere with one another, potentially limiting each others benefits. For example, load-balancing strategies based on response times are to be avoided when a replica control

strategy that bounds the response times is used [Dürango et al., 2014]. This is not only true for brownout, but for every technique that enforces bounded response times [Björkqvist et al., 2018; Björkqvist et al., 2016], like admission control policies [Kihl et al., 2004; Robertson et al., 2003].

In general, the interference between two control policies is a complex problem [Heo and Abdelzaher, 2009; Diaconescu et al., 2017]. Two different decision making strategies, both working well in isolation, can interfere in unpredictable ways with one another, especially when there are delays between the two decisions. For example, the Shortest Queue First (SQF) load-balancing policy has degraded performance when a queue control strategy (like graceful degradation, or admission control) is active at the replica level, as can be seen in the example of Section 2.

We propose a load-balancing and graceful degradation policy that takes into account both the decisions with the advantage of better controlling the response times and the resource utilization of the data center. This paper makes the following contributions:

- It identifies problems with the currently used load-balancing policies, due to the interplay between graceful degradation techniques at the replica level and load balancers that should distribute the load to multiple replicas.
- It proposes a new architecture, with a higher degree of controllability, that includes both load balancing and graceful degradation, solving the mentioned problems.
- It presents the control design for each of the elements in this architecture.
- It validates the proposal with an experimental campaign, comparing it to existing techniques. The proposed architecture outperforms existing ones in terms of predictability and resource usage. It is in fact able to achieve lower variance for the response times, utilizing the data center resources more efficiently.

The paper is organized as follows. Section 2 provides a more precise statement of the problem our solution solves, and details why this is necessary for modern data centers. Section 3 describes our control solution, and shows block diagrams for all the elements involved. It also offers an analysis from the control perspective of the behavior of the cloud platform. Section 4 provides experimental evidence for our claims and shows that the proposed approach is easy to implement and offers competitive advantages in terms of response time management. Section 5 casts the proposed solution in the state of the art, and Section 6 concludes the paper.

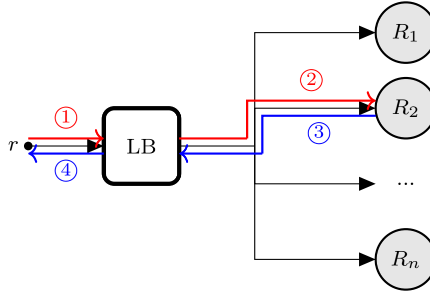


Figure 1. Architecture (one load balancer and multiple replicas) and path of one single request ρ from the user request (step ①) to the response forwarding (step ④).

2. Problem Statement

This paper deals with the problem of designing a load-balancing architecture with graceful degradation. We assume that the architecture is composed of one single load balancer (denoted with LB) and a set of n replicas (denoted with $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The goal of the architecture is to achieve high service predictability. We translate predictability into two related objectives, and measure it in terms of the response times for incoming requests. We want a statistic on the response times (e.g., average, 95th percentile, 99th percentile) to follow a setpoint (a predetermined value, specified for the given cloud application). Also, we want to minimize the variance in response time. A low variance of the worst-case response times, in fact, corresponds to a high degree of predictability. In the remainder of this paper, we assume a setpoint on the 95th percentile of the response times, and use the integrated absolute error (IAE) with respect to this setpoint as our predictability metric. However, similar considerations can be drawn using other statistics.

The path of one single request is shown in Figure 1. We assume that all requests enter the system through one central load balancer (step ①), which in turn routes each request to one of the n replicas (R_2 in the Figure, as shown by step ②). Finally, the replicas serve the requests. Each replica is capable of performing graceful degradation, and thus can choose to serve different amount of content, which requires more or less service time. Here we use brownout [Klein et al., 2014a] for graceful degradation, but other techniques can be applied. Using brownout implies that a request can be served either with or without optional content. The service time used to compute the optional content can be spared, in case the replica detects some capacity shortage. The replica determines the response to the request and communicates it to the load balancer (step ③), which finally replies to the user (step ④). Notice that this is the standard path of a request in a multi-replica archi-

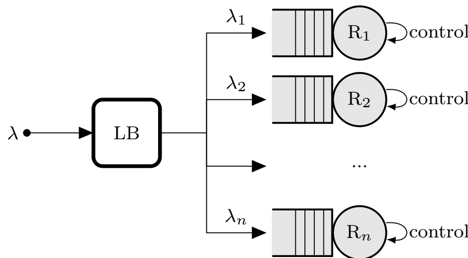


Figure 2. The standard load-balancing architecture. The load balancer routs incoming requests directly to a replica, where the request might spend some time queuing before service. Replicas include graceful degradation controllers.

tecture, used in practical applications and also in earlier research [Dürango et al., 2014; Klein et al., 2014b]. In fact, the replica cannot directly respond to the user, that has queried the server using the IP address of the load balancer. The user would not identify the replica as the server that was queried and would then terminate the connection.

In this architecture, the response produced in step ③ can be used to "piggy-back" information from the replica to the load balancer, without incurring an additional overhead in response time. The load balancer then tears the envelope of the response received by the replica, and only answers to the user with the actual message, in step ④.

The mentioned architecture is commonly implemented as shown in Figure 2. Each replica has an individual queue for requests, and the load balancer routes requests to the queues based on some policy e.g. Round-Robin, SQF, or a weighted probability. In turn, each individual replica has a local graceful degradation strategy — in the brownout case, a response time controller which decides if to serve optional content or not based on the last measured response time from that replica. While this architecture is conceptually simple, the predictability of the response times is highly dependent on the co-design of the load-balancing policy and the controllers in the replicas. The design will also depend on the service discipline used in the replicas (e.g. "First-In First-Out" (FIFO) or Processor Sharing (PS)). In this paper, we will assume a generalized concept of PS being used in the replicas. Specifically, the replicas will serve at a maximum M_C number of requests concurrently from the queue. FIFO and standard PS are then simply the special cases $M_C = 1$ and $M_C = \infty$ respectively. For further details, see [Nylander et al., 2018].

After being routed by the load balancer, requests will spend some nonzero time queuing before service by the replica is started. The average time spent queuing will vary with e.g. workload λ, number of concurrently served

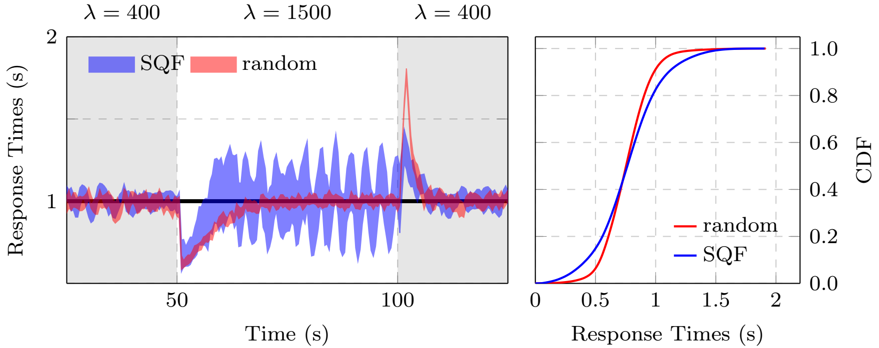


Figure 3. Comparison between random and SQF load-balancing. The left plot shows setpoint and 95% confidence intervals for the 95$^{\text{th}}$ percentile of the response times of the optional-content requests served by a replica. The right plot shows the Cumulative Distribution Function (CDF) of all response times.

requests M_C, etc., and will introduce a delay between the decisions made by the load balancer and the local replica controllers respectively. This is a problem, since delays in between the decisions introduce the risk of routing and graceful degradation counter-acting each other.

Load-balancing policies which have been shown to perform well in case of static service rates can actually counter-act the work of the local controllers in the replicas, leading to poor predictability of response times. This can for example be the case with SQF, despite it being regarded as one of the best load-balancing alternatives [Dürango et al., 2014; Klein et al., 2014b]. An example of this phenomenon is shown in Figure 3. The plots depict the results of an experiment conducted with a simulator[1] that emulates an architecture composed of a load balancer and 5 equal replicas with local graceful degradation controllers, i.e. as in Figure 2, with $n = 5$. The local controllers are using the feedback control strategy from [Nylander et al., 2018], that determines the optional content computation. Each replica in the simulation takes on average 0.014 s to compute the optional content part of the response (with a variance of 0.01s^2), and 0.0002s on average for the mandatory part (with a variance of 0.001s^2). A maximum of $M_C = 15$ requests can be served concurrently in each replica. The run was repeated 20 times, in order to be able to show statistically significant behaviors (using 95% confidence intervals). The simulator uses the open-loop client model and the request arrivals are modeled using the Poisson distribution with arrival rate λ. The simulation

[1] For a description of the simulator used, see Section 4.1.

is split into three different time intervals, in each of them the arrival rate λ is varied. In the time intervals $[0, 50)$ and $[100, 150]$, $\lambda = 400$ and in the time interval $[50, 100)$ $\lambda = 1500$.

The figure compares the SQF load-balancing strategy with a random load balancer. The leftmost plot shows confidence intervals for the 95th percentile of the response times of the requests served with optional content (the critical ones) and their setpoint of 1 s. The rightmost plot shows the Cumulative Distribution Function (CDF) for the two strategies. The use of SQF generates a higher variance in the response times, most notably during the period of heavy workload with $\lambda = 1500$ when requests will spend more time queuing at the replicas. Notably, SQF is performing worse than the simpler random choice policy. Even using specifically “brownout-aware” load-balancing policies [Klein et al., 2014b; Gupta et al., 2007], maintaining predictable response times using the architecture of Figure 2 (the *de facto* standard architecture) remains a challenging task due to the interplay between the different control loops.

To avoid this problem, we instead opt for designing a new architecture where the design of the load-balancing policy and of the local controllers can be done separately, with the aim for them to integrate well from the start. The total response time of a request is divided into two distinct parts: (i) waiting time, and (ii) service time. The load balancer controls the waiting time, and the local controller keeps the service time at a setpoint. In the following section, we describe our proposal, and detail the policies used for both load-balancing and graceful degradation, based on a control-theoretic approach.

3. Proposed Solution

Based on the idea of separating the control of the response times into two distinct parts (one for queueing time and one for pure service time), we propose the load-balancing architecture shown in Figure 4. Contrary to the architecture shown in Figure 2, our proposal contains only one central queue for incoming requests, situated at the load balancer.

The load balancer routes requests from the central queue in a “first come first served” manner. When the load balancer routes a request, a controller decides if the request should be served with optional content (normally) or not (i.e., applying graceful degradation). Based on this decision, the load balancer then attaches a flag to the request and forwards it to the replica with the highest demand for a new request.

In each replica, all the forwarded requests are assumed to be served concurrently. From the implementation perspective, each request is served in a separate thread, and all the threads are run concurrently, sharing the com-

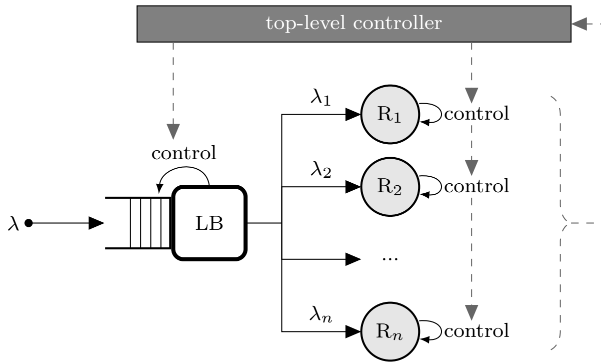


Figure 4. The proposed load-balancing architecture.

putational capacity. At most M_C requests may be served concurrently at each replica. Intuitively, an increase in the number of concurrent requests should result in a longer service time for each of them, and we will use this assumption here.

When a response is produced, a local controller in the replica decides how many more requests it desires to handle, and attaches this integer value to the response. The response is sent back to the load balancer, triggering an event where the attached integer value is used to update a list which keeps track of the current demand of requests from each replica. The load balancer then uses this list to decide where to route the next requests, distributing the requests from to replicas according to their desires.

In summary, a request entering the proposed architecture in Figure 4 goes through the following steps — ⟨LB⟩ indicates that the step is performed by the Load Balancer, ⟨R⟩ that it is performed by the Replica:

1. ⟨LB⟩ The request is put in the queue.
2. ⟨LB⟩ The request waits until it reaches head of the queue.
3. ⟨LB⟩ Routing is triggered with replica demands.
4. ⟨LB⟩ An "optional content" flag is attached to the request.
5. ⟨LB⟩ The request is forwarded to the replica.
6. ⟨R⟩ The request is served by the replica.
7. ⟨R⟩ A response is produced.
8. ⟨R⟩ A new demand value is attached to the response.
9. ⟨LB⟩ The response triggers routing modifications.

10. ⟨LB⟩ The response is sent to the user.

Assuming the time overhead due to routing is negligible, the delay between routing and graceful degradation decisions is now removed. The total response time for a request is separated into: (i) the waiting time in the central queue at the load balancer (step 2), and (ii) the service time in one of the replicas (step 7). The controller in the load balancer decides if optional content should be served or not (step 4), based on a setpoint on the waiting time in the queue (on the time needed to complete step 2). We will refer to this controller as the *waiting time controller*. By flagging a request to be served with optional content or not, the waiting time controller increases or decreases the throughput of the queue, thus affecting the waiting time of future requests.

The local controller in each replica decides how many more requests the replica should demand (step 8). This is based on a setpoint for the service time of requests (for the time needed to complete step 7). We will refer to this controller as the *service time controller*. Each service time controller affects the requests' service time by deciding the number of concurrently served requests and informing the load balancer. The service time setpoint is the same for all replicas, which ensures fairness among the requests.

Finally, we desire the overall infrastructure to follow a global setpoint that prescribes statistics on the response times (e.g., the 95th percentile of the response times of all the replicas should follow a given setpoint). A third controller is then responsible for determining the two setpoints of the other controllers – the setpoint on waiting and service time – dynamically. We refer to the third controller as the *top-level controller*.

In the following, we discuss the design of each of these three controllers in a separate section. Section 3.1 describes the waiting time controller, Section 3.2 details the service time controller, Section 3.3 discusses the top-level controller, and, finally, Section 3.4 describes additional implementational aspects, including our anti-windup strategy.

3.1 Waiting Time Control Design

The waiting time controller is located in the load balancer, and uses the decision of serving optional content or not as an actuator to steer the average waiting time $\bar{t}_w$ to its setpoint $r_{\bar{t}_w}$. Feedback is achieved by directly measuring the waiting time $t_w(\rho)$ of each request ρ right before it is being routed. The controller then attaches a flag, $o(\rho) \in \{0, 1\}$, to the request based on this measurement, where $o(\rho) = 1$ indicates that optional content should be computed and served. For each request ρ, the decision on the value of $o(\rho)$ is based on a threshold ψ_t on the waiting time. The threshold ψ_t is updated periodically, and denoting with k the time interval $[k \cdot t, (k + 1) \cdot t)$, and with $\psi_t(k)$ the value of the threshold in said time interval, the controller behaves

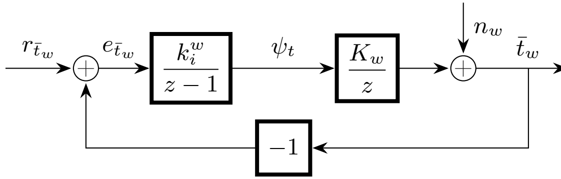


Figure 5. The waiting time control loop design in discrete time.

according to Equation (1). If the measured waiting time is higher than the threshold, then no optional content is served. Otherwise, the request is served with optional content.

$$\begin{array}{lcl} t_w(\rho) > \psi_t(k) & \implies & o(\rho) = 0 \\ t_w(\rho) \leq \psi_t(k) & \implies & o(\rho) = 1 \end{array} \tag{1}$$

In stationarity, the average waiting time $\bar{t}_w$ will stay in the vicinity of the threshold ψ_t. However, the exact relation will depend on the current state of the system. This motivates the need for a feedback controller, which dynamically changes the threshold ψ_t such that $\bar{t}_w$ always follows the setpoint $r_{\bar{t}_w}$. In order to design this controller, a model describing the dynamics from ψ_t to $\bar{t}_w$ is required. As a simplification, if the controller that determines the value of ψ_t is designed to be slow in comparison with the threshold algorithm specified in Equation (1), then $\bar{t}_w$ can be approximated as always staying close to the threshold ψ_t. This is a reasonable approximation, since Equation (1) is very effective at keeping the request waiting times close to the threshold ψ_t, thanks to its event-driven execution. Using this reasoning, the dynamics from ψ_t to $\bar{t}_w$ can be modeled in discrete time as:

$$\bar{t}_w(k+1) = K_w \psi_t(k) + n_w, \tag{2}$$

where K_w is a gain close to 1 and n_w is a stochastic disturbance related to the non-deterministic nature of the arrivals to the load balancer and service times in the replicas. We here use control-theoretical design principles [Åström and Wittenmark, 2011] and compute the *Z-transform* of Equation (2). The pulse transfer function $H_w(z)$ from ψ_t to $\bar{t}_w$ then becomes

$$H_w(z) = \frac{K_w}{z}. \tag{3}$$

In order to achieve zero stationary error with respect to the setpoint $r_{\bar{t}_w}$, integral action is required in the controller. A pure integral controller is here used,

$$C_w(z) = \frac{k_i^w}{z-1}, \tag{4}$$

where k_i^w is the integral gain to be determined. The proposed design for the waiting time control loop is shown in the block diagram in Figure 5.

Closing the loop with the proposed controller leads to the following characteristic equation for the closed loop system:

$$z^2 - z + K_w k_i^w = 0. \tag{5}$$

We desire to place the poles of the closed-loop system system within the unit circle for stability, and on the positive real axis for a desirable transient behavior. This corresponds to the following desired characteristic equation

$$z^2 - (a + b)z + ab = 0, \tag{6}$$

where $0 \leq a$, $b \leq 1$, for the desired locations of the poles. Comparing coefficients in Equations (5) and (6) results in the following system of equations:

$$\begin{aligned} a + b &= 1, \\ K_w k_i^w &= ab. \end{aligned} \tag{7}$$

Simulations suggest that the pole placement $b = 0.92$, $a = 1 - b = 0.08$ gives a good transient behaviour of the closed-loop system, in terms of disturbance rejection and response speed of the controller. Using (7), this implies that we should choose $k_i^w = 0.07/K_w$. Since we expect that $K_w \approx 1$, a reasonable choice for the integrator gain is $k_i^w = 0.07$.

The robustness of this design choice can be tested by using Equation (5) to examine for what values of the process gain K_w the closed-loop system remains asymptotically stable (i.e. when the poles are within the unit circle). Inserting $k_i^w = 0.07$ in (5), the closed loop system remains stable for $K_w \leq 14.3$. Since K_w is expected to have a value close to one, this implies a very robust control design.

An example showing the control action of the waiting time controller when using the proposed architecture during different workloads is presented in Figure 6. The setup is the same as for the comparison made in Figure 3, and the 95% confidence intervals are based on 20 runs. Here we see how the waiting time controller dynamically adjusts the threshold ψ_t with the changing workload such that the mean waiting time $\bar{t}_w$ follows the setpoint r_{t_w}, which has a static value of 0.5 in this example.

3.2 Service Time Control Design

Each replica has a service time controller, responsible for keeping the average service times (for requests serving optional content) $\bar{t}_s$ at the setpoint $r_{\bar{t}_s}$. The value used for feedback is thus the average value of the service times of all completed requests during each time interval k. The service time controller can affect the service times by changing the integer number of simultaneous

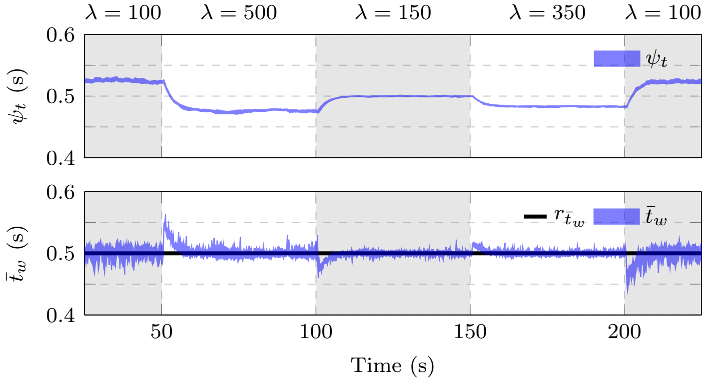


Figure 6. 95% confidence intervals from 20 runs on thresholds ψ_t (upper) and average waiting times $\bar{t}_w$ (lower) using the proposed waiting time controller in the load balancer. The setpoint on the mean waiting time $r_{\bar{t}_w}$ is 0.5.

requests $u_a \in \mathbb{Z}^+$ to run. However, the control signal $u \in \mathbb{R}^+$ computed by the controller is a non-negative real-valued number, which thus has to be quantized as $u_a = \lceil u \rceil$ before it can be actuated (the ceiling function is used here for the quantization).

To be able to assess the behaviour of the control strategy and theoretically analyze the system, we need a model relating u to $\bar{t}_s$. In the modeling process, the quantization effects are neglected, i.e. we assume $u_a = u$. Assuming that all forwarded requests to the replica will be served concurrently, and assuming that a change in u is reflected very fast in $\bar{t}_s$, we can use the following simple discrete-time model:

$$\bar{t}_s(k+1) = K_s u(k) + n_s, \tag{8}$$

where K_s is a gain relating the number of simultaneous requests to the average service times and n_s is a stochastic disturbance describing the variance in the service times. Note that this model (8) has the same structure as the waiting time model (2). As a result, a majority of the analysis in Section 3.1 can be re-used. However, in this case, the gain K_s can not be assumed to always stay close to 1. In fact, K_s is directly related to the speed of the replica, which can vary greatly with time and also be different among the different replicas. This gain thus has to be estimated by the replica controller. The estimation $\hat{K}_s$ is performed, in each replica, using an exponentially weighted moving average filter:

$$\hat{K}_s(k+1) = (1-\alpha)\hat{K}_s(k) + \alpha \frac{\bar{t}_s(k)}{u_a(k)}, \tag{9}$$

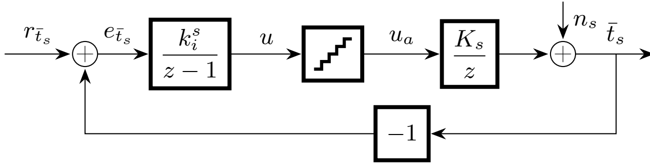


Figure 7. The service time control loop design in discrete time.

where α is a design parameter, here set to $\alpha = 0.5$ (based on preliminary experiments). Using this estimated gain and the controller design in Section 3.1, and in particular the results from Equation (7), the following adaptive integral controller is proposed for the service time:

$$k_i^s = \frac{c(1-c)}{\hat{K}_s}. \tag{10}$$

The location of the slowest closed-loop pole $0 \leq c \leq 1$ is a trade-off between rejection of control errors caused by changes in server speed, robustness to estimation errors in $\hat{K}_s$, quantization errors, and rejection of the stochastic noise n_s. Taking these elements into consideration, we place the pole in $c = 0.8$, which results in a stable closed-loop system as long as $\hat{K}_s \geq K_s/6.25$. We consider this a robust enough design. The adaptive integral control design is thus:

$$k_i^s = \frac{0.16}{\hat{K}_s}. \tag{11}$$

The block diagram of the complete service time model and control design is shown in Figure 7.

The actuation of the quantized control signal u_a, representing the number of simultaneous requests to run in a replica, is as previously mentioned performed using piggy-backing. In more detail, the following steps are involved:

1. At startup, both u and u_a are initialized to zero.

2. The control signal u is updated every time interval k according to the scheme in Figure 7.

3. At every request completion, a new value of u_a is computed: $u_a^{new} = \lceil u \rceil$. The difference $u_a^{'} = u_a^{new} - u_a$ is determined and the old value of u_a is updated to u_a^{new}.

4. The response of the completed request is sent back to the load balancer, using piggy-back to send also $1 + u_a^{'}$, the number of new requests that the replica wants to serve.

The steps above constitute the actuation of u_a, completing the control design. The mentioned design ensures stability, tackles robustness issues, and guarantees a fast convergence, as shown in the experimental validation presented in Section 4.

An example showing the control action and gain estimation of the service time controller when using the proposed architecture is presented in Figure 8. The setup is the same as for the comparison made in Figure 3, but here we instead vary the service times for both optional and mandatory content by scaling them by a factor $1/\mu$ during different time intervals of 50 s. The service time controller is able to efficiently estimate the gain K_s and dynamically adjust the number of concurrently served requests u_a such that the mean service time $\bar{t}_s$ follows the setpoint r_{t_s}, which has a constant value of 0.5 in this example.

3.3 Top-Level Control Design

To ensure that the global setpoint on respose times is followed, we employ a top-level controller. This controller decides the setpoints of the other two controllers, i.e., the setpoint on the waiting and on the service time, respectively $r_{\bar{t}_w}$ and $r_{\bar{t}_s}$. The setpoint r_{t_c} prescribes a statistical measure obtained from the vector of response times, e.g. the 95th percentile. The top-level controller receives the measured value of the same statistic of the response times t_c as a feedback signal. The controller then dynamically adjusts the setpoints $r_{\bar{t}_w}$ and $r_{\bar{t}_s}$.

While the top-level controller should react to persistent errors in the response times, we also wish to avoid being too sensitive to outliers and transient errors in the inner control loops. This motivates the choice of a top-level controller which is slow with respect to the dynamics of the waiting- and service-time control loops. We can then re-use again the analysis from Section 3.1, and propose the following simple integral controller:

$$C_c(z) = \frac{k_i^c}{z-1}. \tag{12}$$

The integral gain k_i^c is chosen as a sufficiently small value. Studying the behavior of the system, we selected $k_i^c = 0.01$.

Using this controller's output signal, we change both the other setpoints simultaneously. We specify a fixed ratio $\gamma \in [0, 1]$, which divides the total response time into a fraction γ (due to the waiting time) and $1 - \gamma$ (due to service time).

A block diagram of our proposed design for the top-level controller is shown in Figure 9. The dashed area in the figure represents the plant to control, while the rest is the top-level controller. In the plant, the inner control loops (Sections 3.1 and 3.2) are represented by the blocks G_{cl}^w and

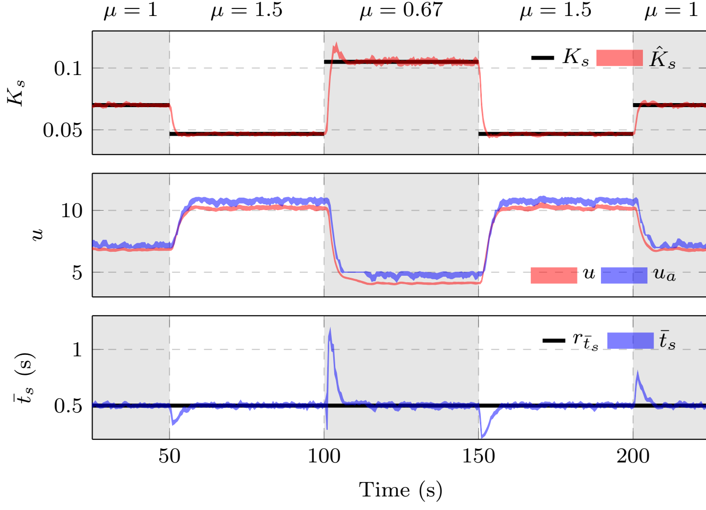


Figure 8. 95% confidence intervals from 20 runs on estimated gain $\hat{K}_s$ (upper), service time control signals u and u_a (middle) and average service times $\bar{t}_s$ (lower) using the proposed service time controller in one replica. The true gain values K_s (upper) and the setpoint on average service time $r_{\bar{t}_s} = 0.5$ (lower) are plotted for reference.

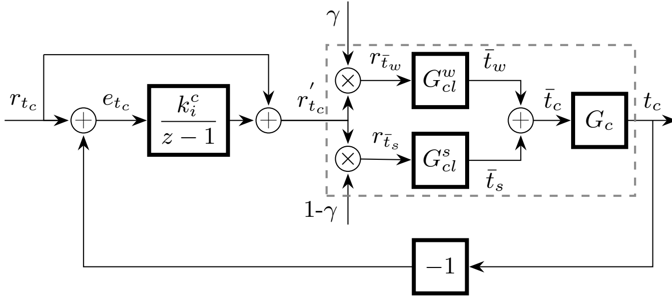


Figure 9. The complete control loop design in discrete time. G_C represents the relation between average response times $\bar{t}_c$ and chosen statistical measure for feedback t_c.

G^s_{cl} for the waiting time- and service time control loop respectively. These control loops are given in detail in Figures 5 and 7. The block G_c represents the conversion block that translates average response times into the statistic that is used as a feedback signal.

In the real system, the top-level controller will be located in the load balancer. From there, updated setpoints on service time can be propagated to the replicas using the requests.

The design parameter γ decides which part of the system the requests will spend most time in, and can be tuned to handle uncertainties in the system. With γ close to one the requests will spend most time waiting in the queue, while the replicas will serve fewer requests concurrently. This is beneficial for the overall predictability of the response times in the case when most uncertainty lies in the service times. The opposite case with γ close to zero is beneficial when most uncertainty lies in the arrival rate of incoming requests.

3.4 Implementation Aspects

The solution proposed in this paper is capable of handling graceful degradation for a wide range of arrival rates. However, it clearly cannot cover all the possible arrival rates, as there are limitations (on the amount of simultaneous requests that can be served in general terms), imposed by the capacity of the replicas. Computing these limitations is fairly straightforward.

If n replicas serve only mandatory content, with a service time of t_m per request, we can compute the upper bound on the overall rate of requests that can be served by the system (with full degradation) as $\mu_{\max} = n/t_m$. In turn, this means that arrival rates $\lambda > \mu_{\max}$ will lead to over-utilization and instability. In this case, it is possible to detect that additional replicas should be started and an auto-scaler can efficiently take care of ensuring a viable operation region. The design of such auto-scaling policy is beyond the scope of this paper. Alternatively, over-utilization can be handled using admission control in the central queue.

During periods of abnormally small workloads, the response times will stay below the setpoint, even though optional content is served to all requests. This poses no issue to the user, but the controllers in the system will see a persistent error in response time, and would ideally like to throttle the throughput further by serving more optional content and more requests concurrently in the replicas. However, since it is not possible to serve more than 100% optional content and route more requests if the central queue is empty, the control signals will be saturated and unable to eliminate the error in response time. Controllers with integral action which experience persistent control errors under saturation are prone to *integrator wind-up*, a well known phenomenon in control theory [Åström and Wittenmark, 2011]. The

effect of integrator wind-up is that the controller will be unresponsive for a period of time when returning to normal workloads, which of course is unacceptable. Being a well-studied problem however, there exists several efficient algorithms in the control literature for removing wind-up from controllers, and the implementation done in our simulator features anti-windup.

Another aspect to consider when implementing strategies for load-balancing and graceful degradation is how the computational time needed to compute the control decisions scale with growing arrival rates and number of replicas. The controllers presented in Section 3 update their decisions based on a fixed sampling period. This means that their computational time is unchanged with respect to the arrival rate. Despite this, some logic has to be executed on a per-request basis (e.g. the decision on optional content, a single comparison of two floating point numbers). The computation that is done per request is in all cases simple, and has negligible execution times. The most expensive computation done on a per-request basis is sorting of the list with number of desired requests for each replica. The time it takes from when a request sends its desired new incoming request value to the time it actually gets forwarded new requests is negligible, and when request are routed to the replica, the corresponding element is removed from the vector that should be sorted. Given the speed of other components in the system, it is unlikely that the list contains demands from more than one replica at any given time, which makes the sorting operation negligible in terms of time complexity.

4. Experimental Validation

This section presents our results. We validate our control strategy using the open source Python-based brownout simulator[2], built to mimic the behavior of cloud applications [Klein et al., 2014a] and described in Section 4.1. We present the results obtained with the new architecture proposal in Section 4.2.

4.1 The simulator

The simulator defines the concepts of *Client*, *Request*, *Replica*, *Replica Controller*, and *Load Balancer*. Clients issue requests to be served by a replica. Clients can behave according to any inter-arrival time distributions and both according to the open-loop or to the closed-loop client model [Schroeder et al., 2006; Alomari and Menasce, 2014]. In the closed-loop model, clients wait for a response and issue a new request only after some think time. In the open loop model, clients do not wait and instead issue new requests with a specific request rate. Being better at modelling a large number of independent users, we performed the evaluation with open-loop clients.

[2] https://github.com/cloud-control/brownout-lb-simulator

Table 1. Bounds on randomized scenario parameters.

Parameter	**Min**	**Max**
n	3	10
t_o $[10^{-2}s]$	1	4
t_m $[10^{-4}s]$	2	8
θ	0.1	0.9
M_C	5	30

For each request, the simulator computes the service time. The time it takes to serve requests with only the mandatory or with the optional content in addition to the mandatory one are computed as random variables, with normal distributions, whose mean and variance are based on profiling data from the execution of experiments on a real machine [Klein et al., 2014a].

Replicas implement a replica controller, that takes care of selecting – for each request – when to serve optional content. In the simulator, we used the replica controller described in [Nylander et al., 2018] and used the suggested tuning parameters. For the control strategy presented in Section 3.2, we use a sampling period of 0.25 s. The controller code developed in the simulator can be directly plugged into brownout-aware applications like RUBiS[3] and RUBBoS[4].

4.2 Experimental Results

To evaluate the predictability of our solution and compare it to the state of the art, we run simulations of 100 randomized scenarios in sequence, each lasting 50s. We then aggregate the results on response times for all the requests in all the scenarios. For the request generation, we use the open-loop client model and the same random seed generator, ensuring that the same number of requests are generated in all the scenarios and that the throughput of the cloud application is the same across the experiments, irrespective of the strategy used.

We use a fixed setpoint $r_{t_c} = 1$s on the 95$^{\text{th}}$ percentile of the response times throughout all scenarios. For each scenario, we randomize the number of replicas n, the average service times t_o (optional content) and t_m (mandatory content) for each individual replica (with the variance fixed to 0.01 s^2 and 0.001s^2 respectively), the number of concurrently running requests M_C (i.e., roughly the number of threads that replicas use to serve requests) and the expected optional content ratio θ.
The values are sampled from uniform probability distributions, with bounds in Table 1.

[3] https://github.com/cloud-control/brownout-rubis
[4] https://github.com/cloud-control/brownout-rubbos

The arrival rate for each scenario is set to

$$\lambda = n\left(\theta \cdot \frac{1}{\bar{t}_o} + (1-\theta) \cdot \frac{1}{\bar{t}_m}\right), \tag{13}$$

where $\bar{t}_o$ and $\bar{t}_m$ are the average service times for optional and mandatory content respectively over the replicas (replicas can be different in their speed). We specify the arrival rate to avoid degenerate scenarios where the system either becomes unstable or where the workload becomes too low – assuming that an auto-scaler is in charge of selecting a correct number of replicas to run in the system.

We compare our proposed solution to three alternative strategies for the same 100 scenarios. For our solution, we use the ratio parameter $\gamma = 0.9$ (i.e. that each request is supposed to spend 90% of its time in the waiting process and 10% of its time being served) as well as $\gamma = 0.7$, as we expect great variations in service times between each scenario. The other evaluated strategies are state of the art solutions from the literature [Nylander et al., 2018; Klein et al., 2014b; Dürango et al., 2014], using the architecture in Figure 2. The evaluated strategies are:

ILAC-γ: The integrated load-balancing and service time control (ILAC) architecture of this paper, with the marked γ parameter. We use both $\gamma = 0.9$ and $\gamma = 0.7$.

Brownout$^{\text{CC}}$ + EPBH: A solution that employs cascaded control, Brownout$^{\text{CC}}$ [Nylander et al., 2018], paired with a brownout-aware weighted probability algorithm for load balancing (EPBH) [Dürango et al., 2014].

Brownout$^{\text{CC}}$ + SQF: The Brownout$^{\text{CC}}$ controller, with the SQF algorithm for load balancing.

Brownout + EPBH: The original brownout controller [Klein et al., 2014a], using the EPBH weighted probability algorithm for load balancing.

To evaluate the predictability of each strategy, we measure the Integrated Absolute Error (IAE) of deviations from the setpoint on the 95$^{\text{th}}$ percentile of response times. Given a sampling interval of length h, we compute the IAE as:

$$\text{IAE} := h \sum_k |r_{t_c}(k) - t_c(k)|, \tag{14}$$

where the summation is done over all sampling intervals k of the experiment. To complement this metric, we also record the standard deviation of the overall response times and the maximum recorded response time.

Table 2. Results from the experiment.

Strategy	**IAE [s]**	**Standard Deviation [s]**	**Max Response Time [s]**
ILAC-0.9	134.4	0.0953	1.41
ILAC-0.7	254.9	0.1412	2.36
Brownout^{CC} + EPBH	423.3	0.2640	2.58
Brownout^{CC} + SQF	823.1	0.2961	3.25
Brownout + EPBH	10980	1.3577	7.27

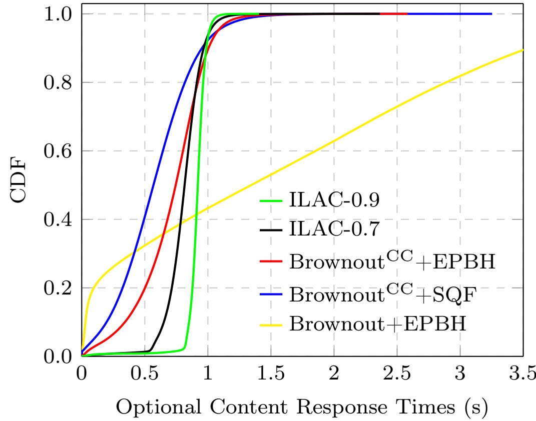


Figure 10. Cumulative Distribution Function (CDF) of response times for all requests with optional content.

The results of the experiment for each strategy are summarized in Table 2, along with Cumulative Distribution Functions of the optional content response times in Figure 10. Comparing the results of ILAC-γ for $\gamma = 0.9$ with $\gamma = 0.7$ indicates that a large value of γ indeed was favourable in the experiment. Still, the ILAC-γ significantly outperforms the other considered strategies in both cases. The closest competitor, Brownout^{CC} combined with EPBH, has a roughly 3 times larger IAE value that ILAC-0.9. The corresponding factor to the Brownout^{CC} + SQF strategy is roughly 6, and over 80 for the Brownout + EPBH strategy. The superior predictability of the proposed ILAC-γ strategy is also reflected in the overall standard deviations and maximum recorded response times, with the maximum response time for ILAC-0.9 being 1.41 seconds. The results show the effectiveness of our proposal and highlight the problem of co-design with the standard architecture in Figure 2, where the efficiency of the EPBH load-balancing alternative varies greatly with the choice of the controller used for graceful degradation.

Table 3. Parameters of 5 selected scenarios (out of the 100 tested).

Scenario	**#1**	**#2**	**#3**	**#4**	**#5**
n	9	6	4	6	9
λ [s^{-1}]	570	890	330	310	570
$\bar{t}_o$ [$10^{-2}s$]	2.5	2.2	2.7	2.3	2.5
$\bar{t}_m$ [$10^{-4}s$]	5.4	4.3	6.3	4.6	5.4
θ	0.62	0.29	0.43	0.84	0.62
M_C	11	13	15	29	11

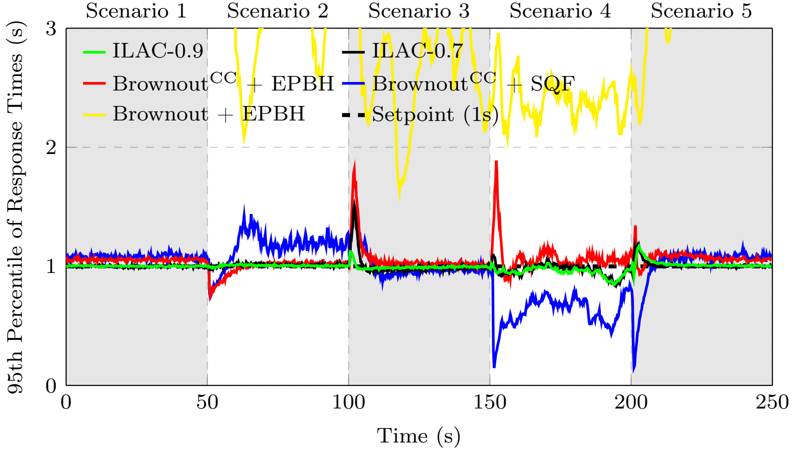


Figure 11. Averaged values of the 95th percentile of response times from 20 runs of 5 selected scenarios. The parameter sets of each scenario is given in Table 3.

The performance of the evaluated strategies are also exemplified in Figure 11, which shows averaged values of the 95th percentile of response times from 20 runs of 5 of the 100 scenarios. The parameter set for each scenario is given in Table 3. We see in in the figure that the performance of the Brownout^{CC} + SQF and Brownout^{CC} + EPBH strategies are heavily dependent on the given scenario, whereas the proposed strategy keeps a high predictability regardless of the parameters used for the simulations. This robustness clearly highlights the benefits of the architecture proposed in Figure 4 combined with a control-theoretical design approach for the decision-making.

5. Related work

Building distributed systems that offer guarantees on their timely execution while the system is subject to *uncertainty* and *changes* is a challenging task. Bounding latencies is of utmost importance, but this is quite difficult in the presence of changes [Wang et al., 2016; Björkqvist et al., 2018; Björkqvist et al., 2016; Javadi and Gandhi, 2017; Ghahremani et al., 2017; Kaler et al., 2017]. Changes are unpredictable, they can be dramatic, and they can include malfunctioning [Iosup et al., 2011], slow down [Fallahi et al., 2013], failures [Guo et al., 2013], and much more. Graceful degradation [Lin and Kulkarni, 2013] is then introduced into the runtime system, to handle these changes and guarantee performance in the presence of uncertainty. This paper shows that graceful degradation and load-balancing can interfere with one another. We focus on a unified solution, to avoid this interference.

In replicated cloud services, load balancers have a crucial role for ensuring resilience and performance [Barroso and Hölzle, 2009; Hamilton, 2007]. Load-balancing algorithms can either be global (inter-data center) or local (intra-data center or cluster-level). Global load-balancing decides what data center to direct a user to, depending on geographic proximity [Lin et al., 2012] or price of energy [Doyle et al., 2013]. Once a data center is selected, a local algorithm directs the request to a machine in the data center. Our contribution is of the local type.

Various local load-balancing algorithms have been proposed. For non-adapting replicas, SQF has been considered very close to optimal, despite it using little information about the state of the replicas [Gupta et al., 2007]. Previous results show that for self-adaptive, brownout replicas, SQF performs quite well [Klein et al., 2014b], but can be outperformed by weight-based, brownout-aware solutions [Dürango et al., 2014]. In this article, we improve on brownout-aware load balancing, by combining the load-balancing strategy with the graceful degradation decision, obtaining better performance in terms of variance of response times, and show improved performance, compared to previously developed algorithms.

6. Conclusion

This paper proposes a new load-balancing architecture that combines the action of the load balancer with graceful degradation techniques like brownout or admission control. We have designed the system and synthesized the load balancing strategies. The advantage of the proposed solution lies in the interplay between the two control solutions. While in previous solutions the two different components – load-balancer and graceful degradation controller – could compete and generate oscillations in response times, our proposal does

not suffer from this issue.

Our proposed architecture has an important tuning parameter: the percentage of time that should be spent waiting and in service for each request. Our experimental campaign showed that – irregardless of the selected percentage time – the response times using the proposed load-balancing strategy are much more predictable than with any other previously explored strategy. Their variance is in fact much smaller than with other strategies, and their maximum is much closer to the desired setpoint than if other strategies are used.

In the future, we plan to combine the proposed architecture with autoscaling features, that trigger new replicas to be started or old replicas to be removed. We also envision using the architecture for fault detection and countermeasures.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Swedish Research Council (VR) for the projects "Feedback Computing" and "Power and temperature control for large-scale computing infrastructures", by the LCCC Linnaeus Center and, by the ELLIIT Excellence Center at Lund University.

References

Alomari, F. and D. A. Menasce (2014). "Efficient response time approximations for multiclass fork and join queues in open and closed queuing networks". *IEEE Trans. Parallel Distrib. Syst.* **25**:6. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.70. URL: http://dx.doi.org/10.1109/TPDS.2013.70.

Åström, K.-J. and B. Wittenmark (2011). *Computer-Controlled Systems*. 3rd ed. Dover Publications Inc., Mineola, NY.

Barroso, L. A. and U. Hölzle (2009). *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool.

Bencomo, N. and A. Belaggoun (2014). "A world full of surprises: bayesian theory of surprise to quantify degrees of uncertainty". In: *36th International Conference on Software Engineering, ICSE14, Companion Proceedings*, pp. 460–463.

Björkqvist, M., R. Birke, and W. Binder (2016). "Resource management of replicated service systems provisioned in the cloud". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, pp. 961–966.

Björkqvist, M., N. Gautam, R. Birke, L. Chen, and W. Binder (2018). "Optimizing for tail sojourn times of cloud clusters". *IEEE Transactions on Cloud Computing* **6**:1, pp. 156–167.

Breitgand, D. and A. Epstein (2012). "Improving consolidation of virtual machines with risk-aware bandwidth oversubscription in compute clouds". In: *Proceedings of the IEEE INFOCOM 2012, Orlando, FL, USA, March 25-30, 2012*, pp. 2861–2865.

Diaconescu, A., K. L. Bellman, L. Esterle, H. Giese, S. Götz, P. R. Lewis, and A. Zisman (2017). "Architectures for collective self-aware computing systems". In: *Self-Aware Computing Systems.* Pp. 191–235.

Ding, S., S. Gollapudi, S. Ieong, K. Kenthapadi, and A. Ntoulas (2011). "Indexing strategies for graceful degradation of search quality". In: *ACM SIGIR conference on Research and development in Information Retrieval.* ACM, pp. 575–584.

Doyle, J., R. Shorten, and D. O'Mahony (2013). "Stratus: load balancing the cloud for carbon emissions control". *TCC* **1**:1. DOI: 10.1109/TCC.2013.4.

Dürango, J., M. Dellkrantz, M. Maggio, C. Klein, A. V. Papadopoulos, F. Hernández-Rodriguez, E. Elmroth, and K.-E. Årzén (2014). "Control-theoretical load-balancing for cloud applications with Brownout". In: *53rd IEEE Conference on Decision and Control, CDC 2014, Los Angeles, CA, USA, December 15-17, 2014*. CDC14, pp. 5320–5327.

Fallahi, N., B. Bonakdarpour, and S. Tixeuil (2013). "Rigorous performance evaluation of self-stabilization using probabilistic model checking". In: *SRDS.* DOI: 10.1109/SRDS.2013.24.

Filieri, A., M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel (2017). "Control strategies for self-adaptive software systems". *TAAS* **11**:4, 24:1–24:31.

Ghahremani, S., H. Giese, and T. Vogel (2017). "Efficient utility-driven self-healing employing adaptation rules for large dynamic architectures". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 590–68.

Greenberg, A., J. Hamilton, D. Maltz, and P. Patel (2008). "The cost of a cloud: research problems in data center networks". *ACM SIGCOMM computer communication review* **39**:1, pp. 68–73.

Grimes, D., D. Mehta, B. O'Sullivan, R. Birke, L. Chen, T. Scherer, and I. Castineiras (2016). "Robust server consolidation: coping with peak demand underestimation". In: *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 271–276.

Grohmann, J., N. Herbst, S. Spinner, and S. Kounev (2017). "Self-tuning resource demand estimation". In: *2017 IEEE International Conference on Autonomic Computing*, pp. 21–26.

Guo, Z., S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou (2013). "Failure recovery: when the cure is worse than the disease". In: *HotOS*, pp. 8–14.

Gupta, V., M. Harchol Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Perform. Eval.* **64**:9-12, pp. 1062–1081.

Hamilton, J. (2007). "On designing and deploying internet-scale services". In: *LISA*. USENIX, 18:1–18:12.

Heo, J. and T. Abdelzaher (2009). "Adaptguard: guarding adaptive systems from instability". In: *Proceedings of the 6th International Conference on Autonomic Computing*, pp. 77–86.

Hoger, M. and O. Kao (2016). "Record skipping in parallel data processing systems". In: *2016 International Conference on Cloud and Autonomic Computing*, pp. 107–110.

Iosup, A., N. Yigitbasi, and D. Epema (2011). "On the performance variability of production cloud services". In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, pp. 104–113.

Javadi, S. A. and A. Gandhi (2017). "DIAL: reducing tail latencies for cloud applications via dynamic interference-aware load balancing". In: *2017 IEEE International Conference on Autonomic Computing*.

Kaler, T., Y. He, and S. Elnikety (2017). "Optimal reissue policies for reducing tail latency". In: *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 195–206.

Kihl, M., A. Robertsson, and B. Wittenmark (2004). "Control theoretic modelling and design of admission control mechanisms for server systems". In: Mitrou, N. et al. (Eds.). *Networking 2004*.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014a). "Brownout: Building more robust cloud applications". In: *36th International Conference on Software Engineering*. ICSE14. ACM, Hyderabad, India, pp. 700–711. ISBN: 978-1-4503-2756-5.

Klein, C., A. V. Papadopoulos, M. Dellkrantz, J. Dürango, M. Maggio, K.-E. Årzén, F. Hernández-Rodriguez, and E. Elmroth (2014b). "Improving cloud service resilience using brownout-aware load-balancing". In: *IEEE 33rd International Symposium on Reliable Distributed Systems*. SRDS14. IEEE Computer Society, pp. 31–40. ISBN: 978-1-4799-5584-8. DOI: 10.1109/SRDS.2014.14. URL: http://dx.doi.org/10.1109/SRDS.2014.14.

Lin, A.-D., C.-S. Li, W. Liao, and H. Franke (2018). "Capacity optimization for resource pooling in virtualized data centers with composable systems". *IEEE Transactions on Parallel and Distributed Systems* **29**:2, pp. 324–337.

Lin, M., Z. Liu, A. Wierman, and L. L. H. Andrew (2012). "Online algorithms for geographical load balancing". In: *IGCC*. IEEE. DOI: 10.1109/IGCC.2012.6322266.

Lin, Y. and S. S. Kulkarni (2013). "Automated multi-graceful degradation: a case study". In: *SRDS*. DOI: 10.1109/SRDS.2013.17.

Litoiu, M., M. Shaw, G. Tamura, N. M. Villegas, H. A. Müller, H. Giese, R. Rouvoy, and É. Rutten (2013). "What can control theory teach us about assurances in self-adaptive software systems?" In: *Software Engineering for Self-Adaptive Systems III. Assurances*, pp. 90–134.

Maggio, M., T. F. Abdelzaher, L. Esterle, H. Giese, J. O. Kephart, O. J. Mengshoel, A. V. Papadopoulos, A. Robertsson, and K. Wolter (2017). "Self-adaptation for individual self-aware computing systems". In: *Self-Aware Computing Systems*. Pp. 375–399.

Neumann, T. (2009). "Query simplification: graceful degradation for join-order optimization". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, pp. 403–414.

Nylander, T., C. Klein, K.-E. Årzén, and M. Maggio (2018). "Brownout^{CC}: Cascaded control for bounding the response times of cloud applications". In: *2018 American Control Conference*. Milwaukee, Wisconsin, USA.

Östberg, P., J. Byrne, P. Casari, P. Eardley, A. Anta, J. Forsman, J. Kennedy, T. L. Duc, M. Marino, R. Loomba, M. Pena, J. Veiga, T. Lynn, V. Mancuso, S. Svorobej, A. Torneus, S. Wesner, P. Willis, and J. Domaschka (2017). "Reliable capacity provisioning for distributed cloud/edge/fog computing applications". In: *2017 European Conference on Networks and Communications (EuCNC)*, pp. 1–6.

Perez, J., R. Birke, and L. Chen (2017). "On the latency-accuracy tradeoff in approximate mapreduce jobs". In: *IEEE Conference on Computer Communications*, pp. 1–9.

Robertson, A., B. Wittenmark, and M. Kihl (2003). "Analysis and design of admission control in web-server systems". In: *Proceedings of the 2003 American Control Conference, 2003.* Vol. 1, 254–259 vol.1.

Schroeder, B., A. Wierman, and M. Harchol-Balter (2006). "Open versus closed: a cautionary tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. USENIX Association, San Jose, CA. URL: `http://dl.acm.org/citation.cfm?id=1267680.1267698`.

Sun, H., R. Birke, W. Binder, M. Björkqvist, and L. Chen (2017). "Accstream: accuracy-aware overload management for stream processing systems". In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 39–48.

Tomás, L. and J. Tordsson (2014). "Cloud service differentiation in overbooked data centers". In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pp. 541–546. ISBN: 978-1-4799-7881-6.

Wang, C., B. Urgaonkar, A. Gupta, L. Chen, R. Birke, and G. Kesidis (2016). "Effective capacity modulation as an explicit control knob for public cloud profitability". In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 95–104.

Xue, J., R. Birke, L. Chen, and E. Smirni (2016). "Managing data center tickets: prediction and active sizing". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 335–346.

Paper IV

Modeling of Request Cloning in Cloud Server Systems using Processor Sharing

Tommi Nylander **Johan Ruuskanen** **Karl-Erik Årzén**
Martina Maggio

Abstract

The interest for studying server systems subject to cloned requests has recently increased. In this paper we present a model that allows us to equivalently represent a system of servers with cloned requests, as a single server. The model is very general, and we show that *no* assumptions on either inter-arrival or service time distributions are required, allowing for, e.g., both heterogeneity and dependencies. Further, we show that the model holds for any queuing discipline. However, we focus our attention on Processor Sharing, as the discipline has not been studied before in this context.

The key requirement that enables us to use the single server G/G/1 model is that the request clones have to receive *synchronized service*. We show examples of server systems fulfilling this requirement. We also use our G/G/1 model to co-design traditional load-balancing algorithms together with cloning strategies, providing well-performing and provably stable designs.

Finally, we also relax the synchronized service requirement and study the effects of non-perfect synchronization. We derive bounds for how common imperfections that occur in practice, such as arrival and cancellation delays, affect the accuracy of our model. We empirically demonstrate that the bounds are tight for small imperfections, and that our co-design method for the popular Join-Shortest-Queue (JSQ) policy can be used even under relaxed synchronization assumptions with small loss in accuracy.

© 2020 ACM/SPEC. Originally published in ACM/SPEC International Conference on Performance Engineering (ICPE), Edmonton, April 2020. Reprinted with permission. The article has been reformatted to fit the current layout.

1. Introduction

In cloud computing, *cloning* is used as a way of speeding up the generation of responses to requests. In this setting, the technique is also known as the generation of *redundant* requests. The basic idea is that, instead of sending requests to only one server, the requests are cloned and sent to multiple servers simultaneously. The response to the request is the result of the server that first completes the processing required to handle the request. When this happens, the pending requests (i.e., the clones that are being processed in the other servers) are cancelled.

Cloning can yield significant improvements to the performance of data centers, as shown in [Ananthanarayanan et al., 2013]. The motivation for cloning comes from the desire to reduce the mean and tail response times of applications running in the cloud. Hosted virtual machines or containers are allocated on shared resources. This means that their behavior is sometimes unpredictable, and the computation times of similar requests can vary among different instances [Dean and Barroso, 2013]. Cloning can thus be viewed as an intuitive way to increase the predictability of cloud applications, by relying on multiple simultaneous copies of a user request. This is the reason why there has recently been an upsurge in the interest for modeling the behaviour of cloud applications subject to cloning.

Existing Results. Cloning is a particular case of the (n, k) fork-join model, where a request is split into n sub-tasks that are distributed to servers. The request completes when at least $k \leq n$ of those task are completed. Cloning implies that the n sub-tasks are identical and $k = 1$. Approximate analysis and latency bounds have been extensively studied for the general (n, k) fork-join systems [Joshi et al., 2012; Shah et al., 2014; Wang et al., 2018], but unfortunately no exact analysis exists when $n \geq 3$. This is, however, not the case for cloning. The first exact analysis of cloning was performed by Gardner et al. [Gardner et al., 2015]. They modeled servers using M/M/1 queues, i.e., queues where the arrivals follow a Poisson process and job service times have an exponential distribution. Other notable contributions concerning cloning with exponential distributions include [Qiu et al., 2016; Gardner et al., 2016b; Ayesta, 2019]. Qiu et al. [Qiu et al., 2016] compares the use of multiple queues (in a distributed servers setting) to a central queue. Gardner et al. [Gardner et al., 2016b] derived results on the largest marginal improvement that can be obtained using the *Redundancy-d* cloning policy, that clones each request to exactly d servers. Ayesta et al. [Ayesta, 2019] improved the analysis of *Redundancy-d*, including different alternatives for handling the request cancellation.

Subsequently, researchers started investigating cloning with specific probability distributions for inter-arrival times and service times, identifying the characteristics of the stochastic (inter-arrival and service time) processes that

make cloning beneficial [Shah et al., 2016]. Joshi et al. [Joshi et al., 2015; Joshi et al., 2017; Joshi, 2018] extended the results obtained with the M/M/1 model to an M/G/1 model, i.e., queues where the arrivals are still determined by a Poisson process, but job service times have a general distribution. However, an underlying assumption for the extension was that all service time distributions are independent and identically distributed (*i.i.d.*), which rules out heterogeneity. This showed that cloning is beneficial if the tail distribution of the service time is log-convex and disruptive if log-concave.

Contribution. In this paper we relax the assumptions made in earlier research contributions. We require *no assumptions* on either inter-arrival or service time distributions, effectively handling heterogeneity. We present a model that is valid with any queuing discipline, however, in this paper we focus on the Processor Sharing (PS) discipline [Kleinrock, 1975]. In fact, to the best of our knowledge, PS has not been studied before in conjunction with request cloning.

The main contributions of this paper are the following:

- We show that the existing equivalent M/G/1 model for cloned systems under i.i.d assumptions can be generalized to allow for *any* inter-arrival or service time distributions, i.e., not requiring the i.i.d assumption. Our G/G/1 model thus allows for both heterogeneous and dependent service time distributions under any queuing discipline, as long as the server system guarantees synchronized service to all request clones. We explore the assumptions that the computing infrastructure needs to fulfill for this to be true.

- For such server systems, we analyze and compute the optimal cloning factor, with respect to the average response times of the server system, for any service time distribution under any load – i.e., the cloning factor that allows us to obtain the lowest possible average latency.

- We analyze more complex server systems, consisting of multiple clusters, and provide a co-design method for joint synthesis of cloning strategy and load-balancing technique. To the best of our knowledge, we present the first provably stable co-designed load-balancing and cloning strategy for the PS discipline.

- We relax the synchronized service assumption and derive bounds for how practical imperfections, such as arrival and cancellation delays, affect the accuracy of our model.

To validate our theoretical findings from a practical standpoint, we built a discrete event simulator with support for request cloning. Our experimental results show that we are able to accurately predict the behaviour of server

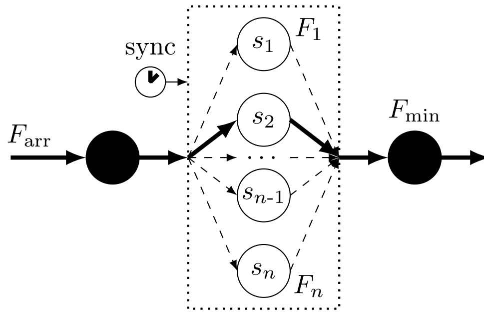


Figure 1. Synchronized service system.

systems subject to cloning. We empirically demonstrate the benefits of co-designing the cloning factor and load-balancing policy, and that the synchronized service assumption can be relaxed for the popular Join-Shortest-Queue (JSQ) policy with small loss in accuracy. Using simulations, we also show that our theoretical bounds can, especially for low arrival and cancellation delays, be used to predict the effect of practical imperfections on our model.

The remainder of the paper is organized as follows. Section 2 presents our model and Section 3 two examples of results that can be obtained. Section 4 shows applications of the model to capture commonly-used data center structures. Section 5 relaxes the synchronized service assumption and shows how this affects our model. Section 6 shows our experimental evaluation. Section 7 presents related research and Section 8 concludes the paper.

2. Synchronized Model

This section formally describes *cloning*, and presents the server system that is the subject of this study. Figure 1 shows a setup example, where n synchronized servers accept requests. An incoming stream of requests is received and each of them is cloned to the n servers. In the most common type of cloning, Cancel-on-Complete cloning, the response to the client is produced by the server that completes the request in the minimum amount of time (s_2 in Figure 1, denoted by thick arrows). The request processing in the other servers is then immediately canceled. In the rest of the paper, we describe the statistical distribution of a random variable X_i using its Cumulative Distribution Function (CDF) and denote the CDF with $F_i(x) = P(X_i \leq x)$. Coherent with this notation, in Figure 1 the CDFs of s_1 and s_n are respectively indicated with F_1 and F_n, the CDF of the request inter-arrival times is indicated with F_{arr}, and the CDF of the minimum service time is marked with F_{min}.

We first define the Cancel-on-Complete cloning approach that we use throughout the paper. Then we discuss synchronized service and the assumptions needed for our theoretical analysis. Finally, we present the main results obtained with our model. Our model holds for any queuing discipline, but we focus our analysis and modeling on the PS discipline due to the lack of prior literature results that properly model this discipline and its closeness to real servers implementations.

DEFINITION 1
(Cancel-on-Complete cloning – CoC cloning) We define Cancel-on-Complete (CoC) cloning as the act of creating n copies of an original request r^o, denoted with $r^c_{1:n}$. More precisely, $r^c_{1:n}$ indicates the vector of n cloned requests. We refer to the i-th request in the vector using the notation r^c_i. We use a similar notation to indicate servers, where $s_{1:n}$ is the vector of servers and s_i indicates the i-th server. The n requests are simultaneously sent to n servers, $s_{1:n}$, on which they eventually enter service. In time, one of the n servers first terminates the computation needed to serve r^o. When this happens, the response that is produced is forwarded to the client and all remaining $n-1$ clones are immediately canceled. □

Another possible cloning approach is *Cancel-on-Start (CoS)*, where all remaining clones get canceled when the first request clone *starts* its service. However, CoS does not apply to the PS discipline as all clones $r^c_{1:n}$ always enter service *immediately*. As a result, we only consider the CoC approach. For the remainder of this paper, we simply use the word cloning to refer to CoC cloning.

We need perfect cancellation to describe the concept of *synchronized service* that forms the basis for enabling our G/G/1 model.

ASSUMPTION 1
(Perfect cancellation) We assume perfect cancellation, i.e. that cancellation of requests takes zero time. □

DEFINITION 2
(Synchronized service) The $r^c_{1:n}$ request clones (sent to servers $s_{1:n}$) receive synchronized service if the clones $r^c_{1:n}$ both enter and leave service simultaneously, i.e., they are dispatched to the servers simultaneously and they are removed from the servers simultaneously at completion of the first clone, implying CoC cloning. This implies the following conditions for the cloning of all original requests r^o:

1. Clones $r^c_{1:n}$ have to be sent simultaneously to all servers $s_{1:n}$.
2. The service in the $n-1$ clones that did not produce a complete response has to be terminated using perfect cancellation, as soon as the fastest server completes the response generation for its clone. □

Note that synchronized service does not imply *immediate* service. Request clones $r^c_{1:n}$ do not have to enter service immediately, and can queue at the servers $s_{1:n}$. Synchronized service only requires that requests enter (and leave) service *simultaneously*. In other words, the synchronized service concept is compatible with any queuing discipline as long as the chosen queuing discipline is the same across all servers $s_{1:n}$. For PS, synchronized service implies that all clones $r^c_{1:n}$ of the same original request r^o experience *identical processor shares*.

The basic setup in Figure 1, with $s_{1:n}$ servers that receive clones, can be the basic block for more complex structures where – for example – a load balancer can be placed in front of multiple of these blocks, each containing a different number of servers, creating a possibly heterogeneous hierarchy. For the remainder of this section, we discuss the basic theoretical concepts using a single block with n servers $s_{1:n}$, as shown in Figure 1. The extension to more complex structures is described in Section 4.

To derive our results, we use the following Theorem, developed in the field of statistics.

THEOREM 1
(Cumulative Distribution Function of the Minimum) Given a set of n random variables $\{X_1, \ldots, X_n\}$ with **any** CDF, and denoting with $F_i(x)$ the CDF of X_i; the CDF of the random variable $X_{\min}$, where $X_{\min} = \min\{X_1, \ldots, X_n\}$ is given by

$$\begin{aligned} F_{\min}(x) \quad = \quad & (-1)^0 \sum_{i=1}^{n} F_i(x) + \\ & (-1)^1 \sum_{i<j} F_{i,j}(x,x) + \\ & (-1)^2 \sum_{i<j<k} F_{i,j,k}(x,x,x) + \ldots + \\ & (-1)^{n-1} F_{i,j,\ldots,n}(x,\cdots,x), \end{aligned} \tag{1}$$

where $F_{i,j}(x,x)$ is the joint CDF of random variables X_i and X_j. If X_i and X_j are independent, i.e. if $F_{i,j}(x,x) = F_i(x)F_j(x)$, Equation (1) reduces to

$$F_{\min}(x) = 1 - \prod_{i=1}^{n} \{1 - F_i(x)\}. \tag{2}$$

□

Proof. This fact is well-known in statistics. The proof uses the inclusion-exclusion principle. A more detailed explanation can be found (for example) in [Modica and Poggiolini, 2012, Proof of Corollary 2.70]. □

Theorem 1 is utilized in the following theorem, which is the main result presented in this section.

Theorem 2
(The Equivalent G/G/1 Model) Assume cloning to a set of n servers using the same queuing discipline with service time distributions $F_{1:n}(x)$ with $x \geq 0$, that guarantee synchronized service. For all original requests r^o arriving with inter-arrival distribution $F_{\text{arr}}(y)$ with $y \geq 0$, the service time of the single request clone that completes service can be equivalently modeled using the distribution of the minimum value $F_{\text{min}}(x)$, determined according to Theorem 1. The server system with cloned requests then behaves equivalently to a G/G/1 server with inter-arrival distribution F_{arr} and service time distribution F_{min}. □

Proof. Each server s_i can be considered as a general and heterogeneous $G/G/1$ queue with inter-arrival distribution $F_{\text{arr}}(y)$ and service time distribution $F_i(x)$. Assume that there exists some $G/G/k$ server model with some inter-arrival distribution $F_{\text{arr}}^{(s)}(y)$ and service time distribution $F^{(s)}(x)$, that governs the response time of requests over the entire system. Synchronized service guarantees that all request clones $r_{1:n}^c$ of an original request r^o enter all servers simultaneously, and that the servers are kept in the same state. Thus the n servers can be seen as a single server of the same queuing discipline with $F_{\text{arr}}^{(s)}(y) = F_{\text{arr}}(y)$. This further implies that the shortest completion time for $r_{1:n}^c$ corresponds to the shortest service time for $r_{1:n}^c$, giving $F^{(s)}(x) = F_{\text{min}}(x)$. Finally, the minimum of n draws from $F_{1:n}(x)$ distributions is equivalent to one draw from $F_{\text{min}}(x)$, thus $k = 1$. □

Theorem 2 allows us to properly model and analyze the service time for server systems with cloned requests.

Remark 1
Theorem 2 does **not** require any assumptions on properties of either the inter-arrival distribution F_{arr} or the service time distributions $F_{1:n}$. Furthermore, the theorem holds for any queuing discipline. □

Compared to previous research effort, the theorem extends the state of the art, in terms of the assumptions needed for its validity. In fact, previous research required to specify properties of either the inter-arrival distribution or the service time distributions. On the contrary, removing the need for these assumptions makes the theorem very general. Using Theorem 2, we define an equivalent G/G/1 model and by that can incorporate in our models both heterogeneity and dependencies across servers. Modeling dependencies across servers allows us to take into account things like the effect of database queries, that are the same no matter which machine is executing the query. We do, however, have to assume synchronized service which implies assumptions that might be unrealistic in practical implementations, such as perfect

cancellations of clones. In Section 5, we study how our model is affected when the synchronized service assumption is relaxed.

3. Examples

In this section, we present two examples of how to use the model described in the previous section. In both examples, we assume synchronized service with n servers under PS.

3.1 Independent Exponential Distributions

The first example describes n heterogeneous servers $s_{1:n}$ whose service times behave according to the same distribution with different parameters. Specifically, we present results obtained with exponentially distributed inter-arrival times y with mean $1/\lambda$. Servers have exponentially distributed service times x with means $1/\mu_i$. Here, we assume that the service time distributions are independent. This is the most common assumption made in all past research and the aim of this example is to verify that we can analytically obtain results covering the most commonly studied case. For example, using queuing theory, Gardner et al. [Gardner et al., 2015] derived results about the distribution of service time of this setup and the FCFS queuing discipline. Here, we show that the same result also applies to any other queuing discipline. The described setup implies

$$\begin{aligned} F_{\text{arr}}(y) &= 1 - e^{-\lambda y}, \\ F_i(x) &= 1 - e^{-\mu_i x}, \end{aligned} \tag{3}$$

with $x \geq 0$, $y \geq 0$. Using Theorem 2, we can model the synchronized service system composed of n servers as a single *equivalent* server having service time distribution $F_{\text{min}}(x)$ as

$$F_{\text{min}}(x) = 1 - \prod_{i=1}^{n} \{1 - F_i(x)\} = 1 - \prod_{i=1}^{n} e^{-\mu_i x} = 1 - e^{-\sum\limits_{i=1}^{n} \mu_i x}. \tag{4}$$

The equivalent single server distribution $F_{\text{min}}(x)$ is thus also exponential, with rate $\mu_{\text{tot}} = \sum_{i=1}^{n} \mu_i$. This means that the n server synchronized service system with cloned requests is *equivalent* to an M/M/1 server with arrival rate λ and service rate μ_{tot}. As anticipated, the expression derived in Equation (4) is the same presented in [Gardner et al., 2015] for the FCFS queuing discipline. However, Theorem 2 allows us to be more general and to show that the same result also holds for any queuing discipline, such as PS, assuming synchronized service.

3.2 Independent Heterogeneous Distributions

In the second example we want to show how to apply the results of Theorem 2 for the case of a synchronized service with n servers having independent and heterogeneous distributions, i.e., where the distribution type changes for each of the servers. We will present a practical example with $n = 3$, assuming that the inter-arrival times y are uniformly distributed between 0 s and 4 s,

$$F_{\text{arr}}(y) = \begin{cases} y/4 & \text{if } 0 \leq y \leq 4.0 \\ 1.0 & \text{if } y > 4.0 \end{cases} . \tag{5}$$

The service time distributions for $s_{1:3}$ are respectively an exponential, a Weibull, and a uniform distribution, with $x \geq 0$ and the following parameters.

$$\begin{aligned} F_1 &= F_{\text{exp}}(x) = 1 - e^{-0.480x} \\ F_2 &= F_{\text{weibull}}(x) = 1 - e^{-0.125x^3} \\ F_3 &= F_{\text{uni}}(x) = \begin{cases} 0 & \text{if } 0 \leq x < 0.5 \\ (x-0.5)/3.5 & \text{if } 0.5 \leq x \leq 3.5 \\ 1 & \text{if } x > 3.5 \end{cases} \end{aligned} \tag{6}$$

We choose these three distributions as they are typically used to model service times. Furthermore, the three distributions have different mean service times (relaxing the assumption of homogeneity, usually made in the literature).

Using Theorem 2, we can compute the equivalent single server service time distribution $F_{\min}(x)$ as follows.

$$\begin{aligned} F_{\min}(x) &= 1 - (1 - F_1)(1 - F_2)(1 - F_3) \\ &= 1 - \{1 - F_{\text{exp}}(x)\}\{1 - F_{\text{weibull}}(x)\}\{1 - F_{\text{uni}}(x)\} \end{aligned}$$

The resulting equivalent model is a G/G/1 model with inter-arrival distribution F_{arr} and service time distribution $F_{\min}$. Figure 2 shows the service time distributions F_1, F_2, and F_3, together with $F_{\min}(x)$.

To demonstrate that the G/G/1 model is in fact equivalent to the cloned server system, we ran 20 simulations with 10^6 requests each, using the simulator described in Section 6. Figure 3 shows the empirical response time CDFs for this example when we simulate both the three servers with cloning case and the equivalent single server case, using the PS discipline. The two response time CDFs are identical, demonstrating the equivalence between the two models.

4. Applications

Here, we use the equivalent G/G/1 model derived in Theorem 2 to analyze different systems under the PS discipline that fulfil the synchronized service

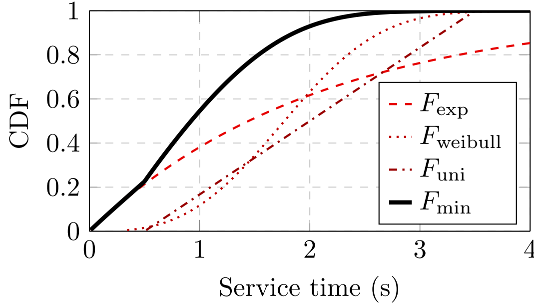


Figure 2. The service time CDFs for the Example with Heterogeneous Servers presented in Section 3.2.

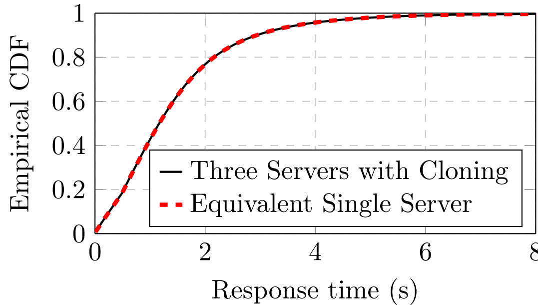


Figure 3. Empirical response time CDFs for the Example with Heterogeneous Servers presented in Section 3.2. Data retrieved through 20 repeated simulations of 10^6 requests each. The 95% confidence intervals lie within the lines.

criterion. The G/G/1 model is compliant with any inter-arrival process F_{arr} and service time distributions $F_{1:n}$, but in order to simplify the analysis, here we restrict ourselves to Poisson arrivals. For the service time distributions, we use the S&Z model, described in Section 4.1.

4.1 S&Z - A Service Time Model

Theorem 2 supports dependencies across service time distributions represented by joint CDFs. However, determining and analyzing these joint distributions is in general difficult. Gardner et al. [Gardner et al., 2016a] propose a model decoupling the task size of the original request r^o, denoted with Z^o, from the server slowdowns affecting clones $r^c_{1:n}$, which we indicate with $S^c_{1:n}$. In our paper, we use the multiplicative version, expressing the service time

$X^c_{1:n}$ for clones $r^c_{1:n}$ as

$$X^c_{1:n} = Z^o \cdot S^c_{1:n}. \tag{7}$$

The idea behind this concept is to model the dependencies across servers $s_{1:n}$ that serve clones $r^c_{1:n}$ of the same original request r^o. As these clones have identical task sizes, it is natural to include the shared task size Z^o in the service time model for clones $r^c_{1:n}$.

The service time model in Equation (7) simplifies our analysis of dependent clones, as the server slowdowns $S_{1:n}$ can be viewed as *independent* across servers and original requests r^o. This allows us to use the simpler expression discussed in Theorem 1 – Equation (2) – when calculating the distribution of the minimum server slowdown $S_{\min}$ for each original request r^o. The complete minimum service time $X_{\min}$ for each r^o is defined as

$$X_{\min} = Z^o \cdot S_{\min}. \tag{8}$$

As shown in Equation (8), $X_{\min}$ belongs to a product distribution. Calculating this complete distribution is difficult, but its first moment can be determined according to:

$$\mathbf{E}[X_{\min}] = \mathbf{E}[Z^o] \cdot \mathbf{E}[S_{\min}]. \tag{9}$$

Exploiting the independence for $S_{1:n}$, we can use Equation (2) to determine the CDF of its minimum distribution $F^S_{\min}$ as

$$F^S_{\min} = 1 - \prod_{i=1}^{n} \left\{1 - F^S_i\right\}, \tag{10}$$

assuming F^S_i known for all S_i. Using $F^S_{\min}$, it is straightforward to determine the first moment of $S_{\min}$. We can then calculate the first moment of $X_{\min}$ using Equation (9), assuming a known task size distribution Z^o. This procedure is used in this section when required by expressions for determining, e.g., response times.

The task sizes Z^o are modeled using a two-phase hyperexponential distribution with balanced means, using $\mathbf{E}[Z^o] = 1/4.7$ and squared coefficient of variation $C^2_{Z^o} = 2$. For the server slowdowns S_i, we use the empirical Dolly distribution, with probability density function defined in Table 1 for the Dolly(1,12) case. First published in [Ananthanarayanan et al., 2013], and later on used in e.g. [Gardner et al., 2016a], the Dolly distribution is based on empirical data on server slowdowns from traces collected from Microsoft Bing's Dryad and Facebook's Hadoop clusters.

In Section 4.2, we derive the results for homogeneous servers, i.e. with server slowdowns S_i from the same Dolly(1,12) distribution.

Table 1. The empirical Dolly(1,12) distribution from [Ananthanarayanan et al., 2013], used to model server slowdowns S.

S	1	2	3	4	5	6	7	8	9	10	11	12
Prob.	0.230	0.140	0.090	0.030	0.080	0.100	0.040	0.140	0.120	0.021	0.007	0.002

4.2 Server Systems

Here, we present two server systems that both fulfill the synchronized service criterion. We investigate, using known expressions in queuing theory, how the cloning factor $c_f \in \mathbf{Z}^+$ (i.e., the number of clones for each request) affects performance and stability.

Clone-to-All. The simplest server system enabling synchronized service is where each request r^o is cloned and clones $r^c_{1:n}$ are sent to all n servers, i.e. with cloning factor $c_f = n$. This system is shown in Figure 1. Using the equivalent G/G/1 model presented in Section 2, we can represent the distribution $F_{\min}$ of service times $X_{\min}$ of this system according to the expression in Theorem 2. This enables us to calculate the mean response times $\mathbf{E}[T]$ of the cloned server system. We use the fact that the mean response time $\mathbf{E}[T^{M/G/1/PS}]$ for the PS queuing discipline only depends on the first moment of the service time distribution G. For our server system under PS, we can thus determine $\mathbf{E}[T^{M/G/1/PS}]$ as

$$\mathbf{E}[T^{M/G/1/PS}] = \frac{\mathbf{E}[X_{\min}]}{1 - \lambda\,\mathbf{E}[X_{\min}]}. \tag{11}$$

For stability, Equation (11) requires the utilization ρ of the cloned server system to be less than 1, thus we get

$$\rho = \lambda\,\mathbf{E}[X_{\min}] < 1. \tag{12}$$

Equations (11)–(12) allow us to exactly determine stability, utilization and mean response time of the Clone-to-All server system for any cloning factor c_f, any arrival rate λ and any service time distribution $F_{\min}$ from which we know the first moment.

Using service times X distributed according to the S&Z model with homogeneous server slowdowns, we can analytically retrieve the first moment of its equivalent service time distribution $X_{\min}$ as described in Section 4.1. By exhaustive search over λ and c_f, we can use Equation (11) to find the optimal cloning factors c_f^{opt} and the corresponding optimal mean response times $\mathbf{E}[T]^{\text{opt}}$, that minimize the mean response times of the server system.

Figure 4 shows an example with exact theoretical results for PS, assuming service times distributed according to the S&Z model described in Section 4.1. The dashed red lines show the optimal cloning factors c_f^{opt}, whereas the blue lines show the corresponding optimal mean response times $\mathbf{E}[T]^{\text{opt}}$. The

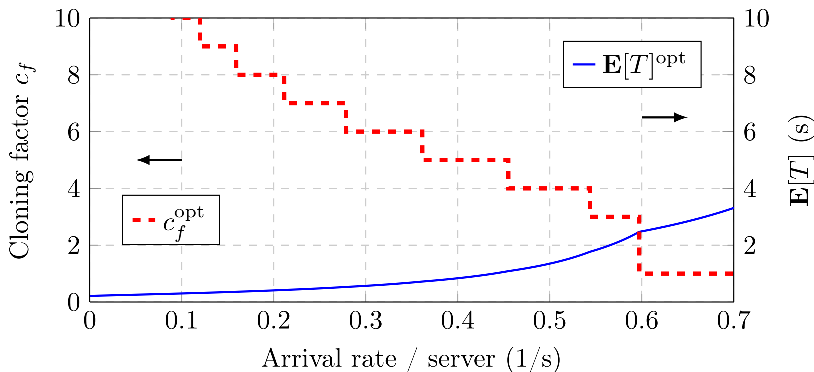


Figure 4. Clone-to-All: Optimal cloning factors together with the corresponding optimal mean response times. Data retrieved from theoretical analysis using Poisson arrivals and S&Z distributed service times.

comparison between cloning factors is performed such that the arrival rate per server is preserved, i.e. if a new server is added λ is scaled accordingly. As expected, higher cloning factors are more beneficial for lower system loads since the clones utilize servers that otherwise would be in idle. A less expected result is that $c_f = 2$ is not optimal for *any* λ for this example. For high system loads, the service time dependencies introduced in the S&Z model, limit the use of cloning and for $\lambda > 0.6s^{-1}$ per server, no cloning ($c_f^{\text{opt}} = 1$) is optimal.

Clone-to-Clusters. The natural extension to only consider cloning to all n servers in the system is to allow for cloning to subsets of servers, as proposed in [Joshi et al., 2017]. To simplify the analysis, we partition the set of n servers into m subsets of equal sizes, containing d servers each, i.e., such that $m \cdot d = n$. We denote these subsets as *clusters*. If an original request r^o is sent to a cluster it gets replicated to all d servers in the cluster. The clusters enable synchronized service for all request clones $r^c_{1:d}$, which means that each cluster can be equivalently represented as a G/G/1 model using Theorem 2.

This cloning strategy allows us to combine an arbitrary load-balancing strategy ℓ that decides to what cluster the original request r^o should be sent, together with the choice of cloning factor $c_f = d$. Figure 5 shows the complete system, that includes the load-balancing strategy ℓ and m clusters.

The choice of the cloning factor c_f depends on the load-balancing strategy ℓ, and we will now show that there is an advantage in conducting a joint design of the two. Given a load-balancing strategy ℓ, the cloning factor $c_f = d$ to be used in each cluster should be chosen such that the mean response time $\mathbf{E}[T]$ for the complete server system is minimized. Varying the value of the

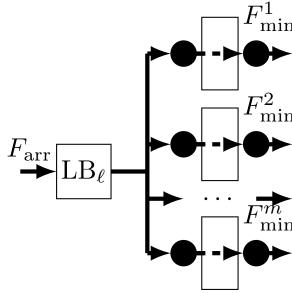


Figure 5. Clone-to-Clusters server system. Each rectangle represents a cluster of d servers that guarantees synchronized service to all clones $r^c_{1:d}$.

cloning factor c_f allows us to design and determine the statistical properties of the behavior of the m clusters (using n available servers), by using the equivalent G/G/1 modeling from Theorem 2 to each cluster separately. Using methods like exhaustive search, it is thus possible to determine the cloning factor c_f and the m clusters that minimize $\mathbf{E}[T]$ under ℓ. An important prerequisite for a successful co-design is that there exists a good enough (possibly approximate) expression for $\mathbf{E}[T]$ (or some other metric) when using the load-balancing strategy ℓ. If ℓ has a well-defined stability criterion, we can co-design a system with a provably stable cloning factor c_f. Here we consider two very common strategies, Random and Join-Shortest-Queue (JSQ).

The Random strategy distributes the original requests uniformly to the servers in the cluster, thus preserving the Poisson properties of the arrival rate λ towards the server system. Each cluster then receives the Poisson arrival rate λ/m. The exact analysis presented in Section 4.2 is directly applicable to the random load-balancing strategy, when deciding the optimal $c_f = d$. We denote this complete co-design as cluster-Redundancy-d (c-R-d), with d representing the cloning factor of each cluster.

The JSQ load balancer always selects the cluster with the shortest queue and sends the cloned requests to that cluster. To co-design the cloning factor with the JSQ strategy, we need to use approximations as no exact results exist for $\mathbf{E}[T]$. As we model our servers using the PS discipline, we utilize the approximation presented in [Gupta et al., 2007]. Exploiting the near-insensitivity towards variability in service time distributions for JSQ under PS, it gives a very good approximation (error within 2-3%) for $\mathbf{E}[T]$, given Poisson arrivals, m clusters and the first moment of the service time distribution. This approximation is thus compatible with the S&Z model. Utilizing this approximation for $\mathbf{E}[T]$ thus allows us to find the optimal cloning factor c_f^{opt}, assuming that the approximations are accurate enough. We denote the complete co-design as cluster-Join-Shortest-Queue-d (c-JSQ-d), where d

Table 2. Theoretical analysis of the co-design example in Section 4.2, using Poisson arrivals and homogeneous service times distributions according to the S&Z model. The arrival rate λ is per server per second.

	$\lambda = 0.30$		$\lambda = 0.38$		$\lambda = 0.52$		$\lambda = 0.62$		$\lambda = 0.70$	
	c_f^{opt}	$\mathbf{E}[T]^{\text{opt}}$(s)	c_f^{opt}	$\mathbf{E}[T]^{\text{opt}}$(s)	c_f^{opt}	$\mathbf{E}[T]^{\text{opt}}$(s)	c_f^{opt}	$\mathbf{E}[T]^{\text{opt}}$(s)	c_f^{opt}	$\mathbf{E}[T]^{\text{opt}}$(s)
c-R-d	6	0.569	6	0.783	4	1.515	1	2.618	1	3.312
c-JSQ-d	6	0.404	4	0.482	3	0.692	2	0.955	1	1.112

represents the cloning factor of each cluster.

Theoretical Co-design Example. To exemplify the co-design procedures of choosing the optimal c_f^{opt}, described in Section 4.2, we study the following example. Assume $n = 12$ servers, i.e. the cluster sizes $d = 1, 2, 3, 4, 6$ and 12 are available. We use Poisson arrivals and homogeneous service times distributed according to the S&Z model, and analyze the optimal cloning factors for each co-design using the exhaustive search method. We consider the queuing discipline PS, and investigate co-designs c-Redundancy-d and c-Join-Shortest-Queue-d. The theoretical results for five selected arrival rates are available in Table 2. In this particular example, the optimal cloning factors c_f^{opt} are equal for both co-designs for $\lambda = 0.30s^{-1}$ and $\lambda = 0.70s^{-1}$, whereas they are different for the other three arrival rates. As the co-design procedure implies optimization criteria that depend on ℓ, c_f^{opt} will in general also depend on ℓ. However, as the example shows, two different co-designs can of course also find the same c_f^{opt} under certain conditions.

For $\lambda = 0.70s^{-1}$ / server, c_f^{opt} is equal to 1 for both co-designs, implying that no cloning is optimal. The fact that we can compare no cloning ($c_f = 1$) to cloning ($c_f > 1$), using the same framework for both results, is powerful as it stops us from recommending cloning when it is not beneficial. Note that for c-JSQ-d the optimal cloning factors are based upon evaluating approximations for $\mathbf{E}[T]$, and their results are thus subject to the quality of the approximations.

5. Non-Synchronized Service

In this section we study the impact of relaxing the assumption of synchronized service, implying that the clones of an original request are no longer guaranteed to receive identical processor shares. First, we consider the effects of non-perfect arrivals and cancellations in the system. This is of high importance as it is most likely impossible to design a perfectly synchronized service in practice. Second, we relax the clone-to-cluster structure to allow for a more general cloning co-design approach, for which we cannot guarantee synchronized service.

In order to analyze our model in a non-synchronized context, the following definitions are needed. Consider a single, specific original request cloned over n servers. Let X_i be the stochastic variable (s.v.) associated with the service time distribution for server s_i. Denote with R_i the runtime of the cloned request on s_i, i.e., its time from service start to departure. Define θ as a vector of n inverse average processor shares for the clones during their runtime, i.e., $\theta_i \geq 1$ states how many requests are present at s_i in average during R_i. Further, let $N = \sum_{i=1}^{n} \theta_i$ be the total average amount of requests in the system during the runtimes of the clones of an original request.

The expected response time of the original request for a specific N can then be written as a function of θ as

$$\mathbf{E}[T|\theta] = \mathbf{E}[\min(\{\theta_j X_j\}_{j=1:n})]. \tag{13}$$

If one assumes that the service time distributions are homogeneous the following interesting result can be obtained.

Theorem 3
The expected response time of an original request cloned to n servers at a specific N is maximized when all elements in θ are equal,

$$\arg\max_{\theta} \mathbf{E}[T|\theta] = \theta^h, \quad \text{where } \theta^h = \{N/n\}_{j=1:n}.$$

Proof. (13) can be rewritten using the Law of Total Expectation

$$\sum_{k=1}^{n} \mathbf{E}[\min(\{\theta_j X_j\}_{j=1:n})|X_k \leq \forall X_i] \cdot \mathbf{P}(X_k \leq \forall X_i), \tag{14}$$

as all X_i belong to the same distribution, $\mathbf{P}(X_k \leq \forall X_i) = 1/n$. Using that the minimum over a set is bounded by all of its members gives

$$\begin{aligned} \mathbf{E}[\min(\{\theta_j X_j\}_{j=1:n})] &\leq \sum_{k=1}^{n} \theta_k \mathbf{E}[X_k|X_k \leq \forall X_i] \frac{1}{n} \\ &= \frac{N}{n} \mathbf{E}[\min(\{X_j\}_{j=1:n})] = \mathbf{E}\left[\min\left(\left\{\theta_j^h X_j\right\}_{j=1:n}\right)\right]. \end{aligned} \tag{15}$$

This proves the theorem. The homogeneous service time distributions yield $\mathbf{E}[X_k|X_k \leq \forall X_i] = \mathbf{E}[\min(\{X_i\}_{i=1:n})]$ for each k. □

Theorem 3 holds under any value of N. For synchronized service, $\theta = \theta^h$ at all times, but in the non-synchronized case this is not true which makes Theorem 3 an important tool for comparison of the two cases.

5.1 Arrival and Cancellation Delays

In real settings, it is highly unlikely that perfect synchronization can be achieved. Instead, imperfections such as slightly different starting times for clones or latency differences between cancelling requests can occur. The imperfections can arise in two stages of the request handling, at arrival and at cancellation, which we model using the notion of *arrival delays* and *cancellation delays.*

Definition 3
Let the arrival delay $a_i \geq 0$ be a s.v. representing the time difference between original request arrival and cloned request arrival on s_i. Further, let the cancellation delay $c_i \geq 0$ be a s.v. representing the time difference between the first completed cloned request on s_k and the departure (cancellation) on s_i. □

We will assume that the distributions of a_i and c_i are independent and homogeneous, we further assume that the service time distributions are homogeneous.

The presence of these imperfections becomes troublesome, as the clone-to-all system can no longer be guaranteed to be synchronized. No synchronization implies that the equivalent G/G/1 model can not be directly applied. It is, however, possible to derive a computable upper bound on the response time of the non-synchronized service. First, the following Lemma is stated.

Lemma 1
Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two, possibly non-synchronized, clone-to-all systems with same number of servers n and arrival rate. If for all N, $\mathbf{E}[T|N, \mathcal{S}_1] \leq \mathbf{E}[T|N, \mathcal{S}_2]$ and $\mathbf{E}[R_i|N, \mathcal{S}_1] \leq \mathbf{E}[R_i|N, \mathcal{S}_2]$, then

$$\mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2]. \tag{16}$$

□

Proof. As $\mathbf{E}[R_i|N, \mathcal{S}_1] \leq \mathbf{E}[R_i|N, \mathcal{S}_2]$ is true for any N, then if $\mathcal{S}_1$ and $\mathcal{S}_2$ are subject to the same arrival rate, the expected number of requests present in the system must be smaller for $\mathcal{S}_1$ than $\mathcal{S}_2$

$$\mathbf{E}[N|\mathcal{S}_1] \leq \mathbf{E}[N|\mathcal{S}_2]. \tag{17}$$

Using the definition of the expected value, (17) can be written as

$$\int Np(N|\mathcal{S}_1)dN \leq \int Np(N|\mathcal{S}_2)dN. \tag{18}$$

This inequality still holds if the function $g(N) = N$ is replaced by two functions that uphold the same inequality for all N, hence

$$\int \mathbf{E}[T|N, \mathcal{S}_1]p(N|\mathcal{S}_1)dN \leq \int \mathbf{E}[T|N, \mathcal{S}_2]p(N|\mathcal{S}_2)dN$$
$$\rightarrow \quad \mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2]. \tag{19}$$

□

It is now possible to compute bounds on the effects of arrival and cancellation delays on the expected response time, by letting $\mathcal{S}_1$ be a clone-to-all system affected by delays and $\mathcal{S}_2$ a synchronized system such that $\mathcal{S}_1$ and $\mathcal{S}_2$ fulfill Lemma 1. Since $\mathcal{S}_2$ is synchronized, the equivalent G/G/1 model can be directly applied to explicitly compute a bound for $\mathcal{S}_1$. We proceed by first considering the two delays separately.

Theorem 4
(Arrival delays) Let $\mathcal{S}_1$ be a clone-to-all system with arrival delay, and $\mathcal{S}_2$ an identical system but synchronized (without delays). Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be subjected to the same arrival rate. Then

$$\mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2] + \mathbf{E}[a]. \tag{20}$$

□

Proof. Consider $\mathcal{S}_1$. For a specific N, the response time of an original request and the runtime of its clones becomes

$$\begin{aligned} T|\theta, \mathcal{S}_1 &= \min\{a_j + \theta_j X_j\}_{j=1:n}, \\ R_i|\theta, \mathcal{S}_1 &= \max(\min\{a_j + \theta_j X_j\}_{j=1:n} - a_i, 0). \end{aligned} \tag{21}$$

The maximum is introduced to prohibit negative R_i when the fastest clone completes before arrival at s_i. The maximum can be dealt with by introducing $b_i = \min(a_i, \min\{a_j + \theta_j X_j\}_{j=1:n})$. By definition, $b_i \leq a_i$ and further, $\min\{a_j + \theta_j X_j\}_{j=1:n} = \min\{b_j + \theta_j X_j\}_{j=1:n}$ as $\theta_j X_j \geq 0$. The following upper bound can then be created

$$\begin{aligned} R_i|\theta, \mathcal{S}_1 &\leq \max(\min\{b_j + \theta_j X_j\}_{j=1:n} - b_i, 0) \\ &= \min\{b_j + \theta_j X_j\}_{j=1:n} - b_i. \end{aligned} \tag{22}$$

The expected runtime is bounded as

$$\mathbf{E}[R_i|\theta, \mathcal{S}_1] \leq \mathbf{E}\left[\min\{b_j + \theta_j X_j\}_{j=1:n}\right] - \mathbf{E}\left[b\right]. \tag{23}$$

Following the proof of Theorem 3, we can state that

$$\begin{aligned} \mathbf{E}[\min(\{b_j + \theta_j X_j\}_{j=1:n})] &\leq \sum_{k=1}^{n} \left(\mathbf{E}[b_k + \theta_k X_k | X_k \leq \forall s_i]\right) \frac{1}{n} \\ &= \mathbf{E}[b] + \mathbf{E}[\min\{\theta_j^h X_j\}_{j=1:n}], \end{aligned} \tag{24}$$

which gives

$$\begin{aligned}\mathbf{E}[R_i|\theta,\mathcal{S}_1] &\leq \mathbf{E}[\min\{\theta_j^h X_j\}_{j=1:n}] + \mathbf{E}[b] - \mathbf{E}[b] \\ &= \mathbf{E}[\min\{\theta_j^h X_j\}_{j=1:n}] = \mathbf{E}[R_i|\theta^h,\mathcal{S}_2].\end{aligned} \tag{25}$$

Further, using Eq. (24) the response time for a request under a specific N can be bounded as

$$\begin{aligned}&\mathbf{E}[T|\theta,\mathcal{S}_1] = \mathbf{E}[\min\{a_j + \theta_j X_j\}_{j=1:n}] \\ &\leq \mathbf{E}[\min\{\theta_j^h X_j\}_{j=1:n}] + \mathbf{E}[a] = \mathbf{E}[T|\theta^h,\mathcal{S}_2] + \mathbf{E}[a].\end{aligned} \tag{26}$$

As the two inequalities Eq. (25) and (26) hold for all $\theta \in \{\sum \theta_i = N,\ \theta_i \geq 1\ \forall i\}$, the statements can be conditioned on N instead without loosing the inequality property. Then using Lemma 1 the original statement is proven.□

Note that for Theorem 4, $\mathbf{E}[a]$ does not affect the stability of $\mathcal{S}_2$. Thus if $\mathcal{S}_2$ is stable, then so is $\mathcal{S}_1$ regardless of arrival delays.

THEOREM 5
(Cancellation delays) Let $\mathcal{S}_1$ be a clone-to-all system with cancellation delays, and $\mathcal{S}_2$ an identical system but synchronized (without delays) and with service time $X_i|\mathcal{S}_2 = X_i|\mathcal{S}_1 + \mathbf{E}[c]$. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be subject to the same arrival rate. Then

$$\mathbf{E}[T|\mathcal{S}_1] \leq \mathbf{E}[T|\mathcal{S}_2]. \tag{27}$$

□

Proof. Consider $\mathcal{S}_1$. For a specific N, the response time of an original request and the runtime of its clones become

$$\begin{aligned}T|\theta,\mathcal{S}_1 &= \min\{\theta_j X_j\}_{j=1:n}, \\ R_i|\theta,\mathcal{S}_1 &= \min\{\theta_j X_j\}_{j=1:n} + \min(c_i, \theta_i X_i - \min\{\theta_j X_j\}_{j=1:n}).\end{aligned} \tag{28}$$

The runtime incorporates the chance that a cloned request completes after $\min\{\theta_j X_j\}_{j=1:n}$ but before c_i has passed, thus the extra minimum. Further, the response time for a specific N is unaffected by the cancellation delay. As the response time is longer if the service time is longer, it can be trivially stated that

$$\mathbf{E}[T|\theta,\mathcal{S}_1] \leq \mathbf{E}[T|\theta^h,\mathcal{S}_2]. \tag{29}$$

The runtime for each server is thus bounded by the cancellation delay

$$R_i|\theta,\mathcal{S}_1 \leq \min\{\theta_j X_j\}_{j=1:n} + c_i. \tag{30}$$

The expected runtime for each server can thus be bounded as

$$\begin{aligned}\mathbf{E}[R_i|\theta,\mathcal{S}_1] &\le \mathbf{E}[\min\{\theta_j X_j\}_{j=1:n} + c_i] \\ &\le \mathbf{E}[\min\{\theta_j (X_j + \mathbf{E}[c])\}_{j=1:n}] \\ &\le \mathbf{E}[R_i|\theta^h,\mathcal{S}_2], \end{aligned} \tag{31}$$

using Theorem 3 in the last step. As the two inequalities Eq. (29) and (31) hold for all $\theta \in \{\sum \theta_i = N,\ \theta_i \ge 1\ \forall i\}$, the statements can be conditioned on N instead without losing the inequality property. Then using Lemma 1 the original statement is proven. □

Note that for Theorem 5, for large $\mathbf{E}[c]$ the upper bound can become infinite despite potential stability of $\mathcal{S}_1$. Thus the arrival rate of the system has to be less than $1/(\mathbf{E}[X_i] + \mathbf{E}[c])$ to guarantee stability for both $\mathcal{S}_1$ and $\mathcal{S}_2$. The following Theorem shows that the effect of both arrival and cancellation delays can be bounded by the sum of the individual bounds.

THEOREM 6
(Combined delays) Let $\mathcal{S}_1$ be a clone-to-all system with both arrival and cancellation delays, and $\mathcal{S}_2$ an identical system but without delays and thus synchronized and $X_i|\mathcal{S}_2 = X_i|\mathcal{S}_1 + \mathbf{E}[c]$. Let both systems be subject to the same arrival rate. Then

$$\mathbf{E}[T|\mathcal{S}_1] \le \mathbf{E}[a] + \mathbf{E}[T|\mathcal{S}_2]. \tag{32}$$

□

Proof. Consider $\mathcal{S}_1$. For a specific N, the response time is still unaffected by the cancellation delay and the runtime for each clone becomes a clear combination of the two separate delay cases,

$$\begin{aligned} T|\theta,\mathcal{S}_1 &= \min\{a_j + \theta_j X_j\}_{j=1:n}, \\ R_i|\theta,\mathcal{S}_1 &= \max\Big(\min\{a_j + \theta_j X_j\}_{j=1:n} - a_i \\ &\qquad + \min(c_i, a_i + \theta_i X_i - \min\{a_j + \theta_j X_j\}_{j=1:n}), 0\Big). \end{aligned} \tag{33}$$

Following the proofs to Theorems 4 and 5 with the previous two equations, it is easy to see that the two bounds are additive. □

The benefit of Theorems 4-6 is two fold. First, they show that small imperfections are not detrimental when trying to implement synchronized service in practice. Further, the bounds are computable given that the expected response time of the equivalent G/G/1 model is computable, which gives a way of making informed decisions in capacity planning of such systems. However, the Theorems are only strictly valid if one assumes that a_i, c_i, X_i are homogeneous and known, which is not the case for all systems.

5.2 Clone-to-Any

The co-design procedure denoted c-ℓ-d in Section 4.2, assuming a clone-to-clusters structure, is of interest as it provides a way to compute and quantify the performance of such systems. The design itself is, however, not that intuitive as it is superfluous to pre-partition the servers into clusters. In practice, a more natural approach would instead be to allow the load-balancing strategy ℓ to, for each original request, choose c_f unique servers from $s_{1:n}$ to clone to. We define this as the *clone-to-any* cloning strategy, and denote co-designs under clone-to-any as a-ℓ-d for cloning factor $c_f = d$ and load-balancing strategy ℓ.

For a-ℓ-d co-designs, synchronized service is no longer guaranteed which implies that the equivalent G/G/1 model is not directly applicable. A measure that quantifies this non-perfect synchronization is the *clone error* ϵ, defined for clones $r^c_{1:d}$ to an original request r^o as $\epsilon = \mathbf{D}[p^c_{1:d}]/\mathbf{E}[p^c_{1:d}]$, with $\mathbf{D}$ the standard deviation and $p^c_{1:d}$ the processor shares for $r^c_{1:d}$. For $\mathbf{E}[\epsilon] > 0$, the system is non-synchronized, for $\mathbf{E}[\epsilon] = 0$ it is synchronized and for small $\mathbf{E}[\epsilon]$ we have *near-synchronization.*

It is intuitive to believe that c-ℓ-d could be used to form an adequate approximation of a-ℓ-d. In fact, the less utilization ρ a system under a-ℓ-d is subject to, the more similar to c-ℓ-d it becomes. This is formalized in the following theorem.

THEOREM 7
Consider the two server systems $\mathcal{S}_1$ and $\mathcal{S}_2$, where $\mathcal{S}_1$ uses a-ℓ-d and $\mathcal{S}_2$ c-ℓ-d but otherwise are identical.

$$\text{Then as } \rho \to 0, \quad \mathbf{E}[T|\mathcal{S}_1] \to \mathbf{E}[T|\mathcal{S}_2]. \tag{34}$$

□

Proof. The smaller the utilization, the larger the probability that all clones $r^c_{1:d}$ to an original request r^o execute alone on their servers. As $\rho \to 0$ then processor shares $p^c_i \to 1$ for all r^c_i. If $p^c_i = 1$ for all r^c_i, then $\mathbf{E}[\epsilon] = 0$ and the clones are synchronized. □

As small $\mathbf{E}[\epsilon]$ implies near-synchronization, it is using c-ℓ-d possible to derive an accurate approximation for the mean response time of a-ℓ-d under low loads, for any ℓ and c_f. For more general system loads ρ, the similarity between a-ℓ-d and c-ℓ-d depends on the choice of ℓ. In particular, if ℓ is good at keeping p_i similar for all clones to the same original request, a-ℓ-d will behave close to synchronized as $\mathbf{E}[\epsilon]$ will be small. In Section 6, the load-balancing strategies random and JSQ, with clone-to-any co-designs denoted a-R-d and a-JSQ-d, are compared to their c-ℓ-d counterparts.

6. Evaluation

In this section, we demonstrate and evaluate the examples and claims stated in the two previous sections, using our own discrete-event simulator[1]. We refrained from using existing simulators like CloudSim [Calheiros et al., 2011], because our evaluation requires us to simulate the cloud application-level behavior and not only the infrastructure behavior. For this reason, we took inspiration from the brownout [Klein et al., 2014] simulator[2] and modified it to remove the adaptation layer and added cloning functionality. In the simulator we include the options to define: (i) the inter-arrival time distribution $F_{\text{arr}}(x)$, (ii) the service time distributions $F_{1:n}(x)$ for our n servers, (iii) the cloning factor c_f, (iv) the load balancing strategy and (v) arrival and cancellation delays. The arrival and service time distributions can be heterogeneous and we allow the user to set them based on empirical CDF data. All simulations in this section are run using the PS discipline.

Our artifacts are available at Zenodo[3]. They consist of our simulator code complete with the scripts that we ran for our simulations. Furthermore, the artifacts provide more details on the simulator architecture, and instructions on how to reproduce our results.

6.1 Server Systems

All experiments in this subsection are evaluated over 20 independent simulations per scenario with unique random seeds, each with 10^6 incoming requests from Poisson arrivals. The service time distribution is the S&Z model as described in Section 4.1.

Clone-to-All. The clone-to-all system in Section 4.2, for which the G/G/1 model yields an exact analysis for c_f^{opt} and $\mathbf{E}[T]^{\text{opt}}$, was simulated with a sweep over the arrival rates. The 95% confidence intervals for the results of the simulations are shown together with the theoretical values in Figure 6. As can clearly be seen, the simulated c_f^{opt} and $\mathbf{E}[T]^{\text{opt}}$ follow their theoretical values closely.

Theoretical Co-designs. We further evaluate the co-designs presented in Section 4.2 using the simulator. The results are shown as 95% confidence intervals in Figure 7, plotted together with the theoretical values for both co-designs c-R-d and c-JSQ-d. For the optimal clone factor c_f^{opt}, the simulated and theoretical values match perfectly. The same applies for the matching of $\mathbf{E}[T]$, at least for c-R-d where the theoretical values are obtained via exact analysis. In fact, the simulated c-JSQ-d mean response times are slightly (1-3%) off compared to the theoretical values. The reason is that the JSQ values

[1] https://github.com/tomminylander/cloning-simulator
[2] https://github.com/cloud-control/brownout-simulator
[3] https://doi.org/10.5281/zenodo.3635905

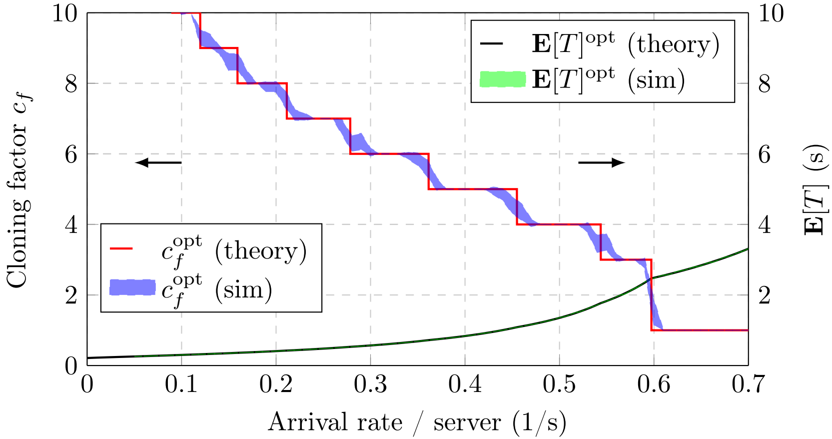


Figure 6. Clone-to-All: Comparison of theoretical values with 95% confidence intervals for the simulation results.

are based on (highly accurate) approximations. As our co-designs succeed in finding all optimal cloning factors c_f^{opt}, the simulations suggest that the co-designs perform well even when the accuracies of the involved approximations of $\mathbf{E}[T]$ are not perfect.

6.2 Non-Synchronized Service

In order to perform a general evaluation, all experiments in this subsection are evaluated over 1000 randomized scenarios with unique random seeds, each with 10^6 incoming requests from Poisson arrivals. We use the following service time distributions: (i) S&Z model from Section 4.1, (ii) Exponential ($\mu = 1$), (iii) Weibull (shape=0.5, scale=0.5), (iv) Pareto (Type 1, shape=2.5, scale=0.6) and (v) Uniform ($X_i \in [0, 2]$). The mean service time $\mathbf{E}[X]$ of all the above distributions at cloning factor $c_f = 1$ is 1s. The utilizations considered are $\rho^{sim} = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]$. For each scenario, we randomly select a service time distribution from (i)-(v), a utilization from ρ^{sim}, a number of servers from s_n^{sim} and a cloning factor from c_f^{sim}. The two latter are defined below.

Arrival and Cancellation Delays. First, we evaluate the theoretical bounds from Section 5.1 for clone-to-all server systems. We use $c_f^{sim} = s_n^{sim} = [2, 3, 4, 5, 6, 7, 9, 10]$. Additionally, from $delay^{sim} = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5] \cdot E[X]$, we randomly select a normalized mean delay. We run three separate experiments: (a) arrival delays only, (b) cancellation delays only and (c) both delays present. In the latter experiment, the mean delays for each scenario are chosen such that, for $0 \leq \gamma \leq 1$ uniformly random,

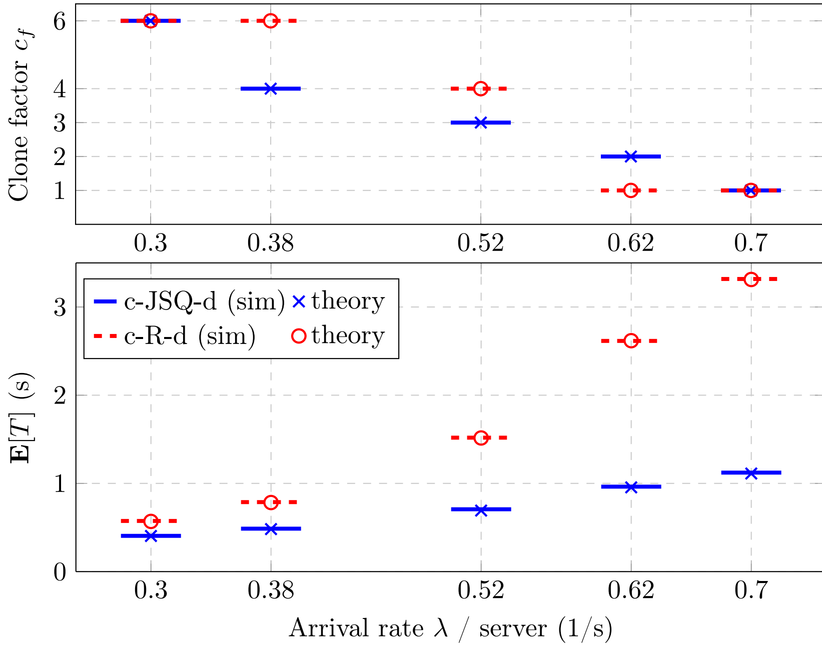


Figure 7. Co-designs: Comparison of theoretical values with 95% confidence intervals for the simulation results. The legend applies to both figures.

$\gamma\mathbf{E}[a] + (1-\gamma)\mathbf{E}[c]$ becomes the chosen delay from $delay^{sim}$. The results are available in Figure 8, and show that for low normalized delays (0.01-0.05) all normalized $\mathbf{E}[T]$ are close to 1. This implies that the synchronized model describes these cases accurately. Further, for the low delays the bounds are tight in all three experiments, which implies that the bounds are very useful for these delays. However, for the higher delays the normalized $\mathbf{E}[T]$ is larger. Also, the bounds for the cancellation delay become very large, and for some scenarios with delay 0.2 and 0.5 they even become infinite as the bound can not guarantee stability for these cases. As none of our simulated scenarios were unstable, it is obvious that the cancellation delay bound has limited usage for these higher delays.

Clone-to-Any. Second, the clone-to-any co-designs are compared to their synchronized counterparts. Here we set $s_n^{sim} = [4, 6, 9, 12, 15, 21, 27, 30, 45, 48]$ and randomly choose c_f, such that for each scenario the chosen number of servers is evenly divisible by c_f.

Figure 9 shows the results for both random and JSQ. The upper plot shows the mean clone errors $\mathbf{E}[\epsilon]$, and it can clearly be seen that for a-JSQ-d the values are much smaller than for a-R-d. This implies that a-JSQ-d is a

much better approximation of its synchronized counterpart than a-R-d. The lower plot, showing normalized $\mathbf{E}[T]$, confirms this statement as the values for a-JSQ-d are much closer to 1. For low ρ, both co-designs approximate the synchronized behavior well in accordance with Theorem 7. For a-JSQ-d, the normalized $\mathbf{E}[T]$ are very close to 1 for all utilizations suggesting that we have near-synchronized service regardless of the arrival rate. This property can be intuitively explained by looking into the JSQ algorithm. As JSQ always sends the clones to the servers with the least amount of running requests, this will cause the servers to always have very similar amounts of running requests. The clones $r^c_{1:d}$ of the same original request r^o will then always receive very similar processor shares, leading to small mean clone errors $\mathbf{E}[\epsilon]$.

Looking more closely at the normalized $\mathbf{E}[T]$, it can be observed that the values for a-JSQ-d and a-R-d never exceed 1. As our simulation study is fairly general, considering many different parameters, this suggests that the mean response times for the synchronized c-ℓ-d co-design might actually form an upper bound for the a-ℓ-d counterparts. This claim is partially supported by Theorem 3, which can be read as that synchronized service in fact always is worse than non-synchronized. However, we have not been able to finalize the proof to hold for complete co-designs, and it will thus have to be left for future work.

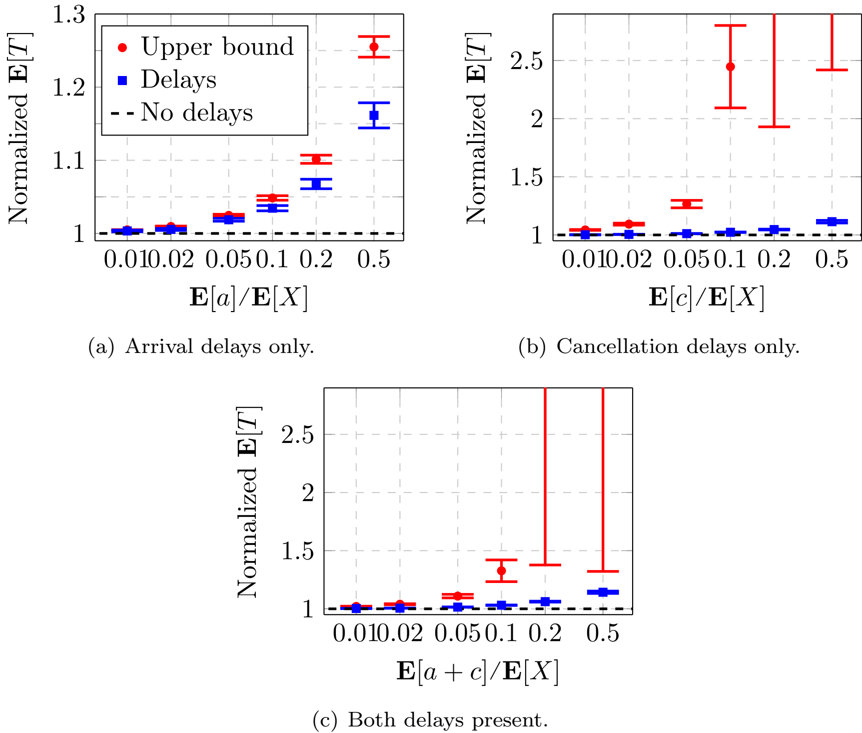
6.3 Summary

Our simulation campaign shows good compliance with our theoretical findings. For both the clone-to-all plots in Figure 6, and the co-design plot in Figure 7, our model predicts the optimal cloning factors c_f^{opt} with high accuracy. Figure 8 shows that, especially for low arrival and cancellation delays, our theoretical bounds can be used to predict the effect of practical imperfections on our model. Figure 9 shows the interesting near-synchronized service property of the JSQ policy, suggesting that our model could accurately describe setups involving the JSQ load-balancer, where synchronized service is not guaranteed.

7. Related Work

Cloning has been studied in the research literature, although in most of the cases previous studies were limited to exponential distributions for service times and the FCFS discipline. We briefly presented an overview of the cloning literature in the introduction of this paper. Here we provide additional details and comparisons with the most related research contributions.

In contrast to pure cloning, speculative execution [Ananthanarayanan et al., 2010; Zaharia et al., 2008] has previously been studied to remedy the effects of slow tasks in large data frameworks such as MapReduce [Dean and



(a) Arrival delays only.

(b) Cancellation delays only.

(c) Both delays present.

Figure 8. Arrival and cancellation delay simulations. The normalization of $\mathbf{E}[T]$ is performed such that each value is divided by the theoretical value without delays. The intervals represent 95% confidence intervals. The legend applies to all figures.

Ghemawat, 2008] or Spark [Zaharia et al., 2012]. Using speculative execution, the infrastructure keeps track of request handling progress and launches copies of slow tasks to reduce the total execution time. As explained by Ganesh et al. [Ananthanarayanan et al., 2013], cloning can be viewed as an extreme case with no speculation time.

Restricting the arrival and service time distributions to being exponential and the queuing discipline to FCFS, the method presented in [Gardner et al., 2015] is able to handle cases that are not covered by the synchronized service definition, e.g., to simultaneously handle both cloning and non-cloning request classes. However, as a result of Theorem 2, here we show that the result presented in [Gardner et al., 2015] is valid also — in the case of server systems with synchronized service — for other queuing disciplines, including for example PS.

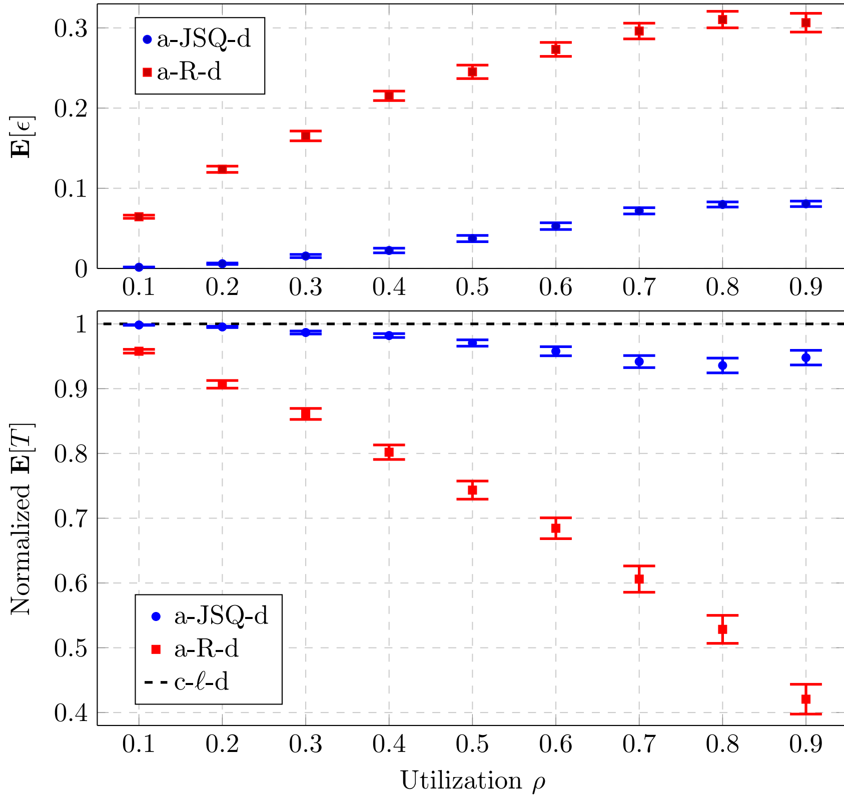


Figure 9. Simulations comparing a-ℓ-d to c-ℓ-d for random and JSQ. The normalization of $\mathbf{E}[T]$ is performed such that each value is divided by the value for the c-ℓ-d counterpart. The intervals represent 95% confidence intervals.

Gardner et al. [Gardner et al., 2016a] propose a service time model decoupling task sizes from server slowdowns, conveniently allowing for modeling of dependencies between request clones. We use this model throughout our paper (denoting it as the S&Z model), and show that using our G/G/1 modeling concept, its statistical properties under cloning can be analyzed even under PS. In the same paper [Gardner et al., 2016a], the authors propose the cloning strategy Redundant-to-Idle-Queue (RIQ), a policy that clones requests to all idle servers that it finds. The RIQ strategy is both provably stable and analytically tractable within the S&Z model, but cannot detect scenarios where cloning actually deteriorates performance. Our co-design procedure presented in Section 4 is, on the contrary, able to identify scenarios where cloning is not beneficial and should be avoided. In these scenarios, the

optimal cloning factor is equal to 1.

It is possible to determine guidelines for cloning factors for service time distributions with specific properties [Shah et al., 2016]. The results presented in [Shah et al., 2016] are applicable to arbitrary arrival processes, and examples where cloning is beneficial include i.i.d. memoryless service time distributions. In our paper, we go beyond this and utilize the G/G/1 modeling in Section 2 to determine optimal cloning factors for any service time distributions. However, we do require server systems to guarantee synchronized service.

In an attempt to make cloning models closer to real implementations, Lee et al. [Lee et al., 2017] worked on modeling and analyzing the overhead of cancellations, and the effects on the optimal scheduling policy. We also consider practical imperfections in our paper, but we instead focus on studying the accuracy of our synchronized model when subjected to e.g. arrival and cancellation delays.

Joshi et al. [Joshi et al., 2017] show that an $(n,1)$ fork-join system can be equivalently represented by an M/G/1 queue, under i.i.d assumptions for service time distributions. Utilizing Theorem 1 and 2, we show that, under synchronized service, a server system under cloning can equivalently be represented by a G/G/1 model, without any assumptions on either interarrival or service time distribution. A group-based random cloning policy is presented in [Joshi et al., 2017], that roughly corresponds to our cluster co-design with ℓ as the random load-balancing algorithm. Our G/G/1 modeling holds for any queuing discipline, allowing us to present and analyze a wider class of cloning co-designs. Specifically, we are able to co-design the JSQ policy together with the cloning factor for the PS discipline. Joshi et al. [Joshi et al., 2017] show that for log-convex tail distributions, cloning to all n servers is optimal even in the heavy traffic regime. Additionally, the paper also takes the computing cost into account when deciding cloning factors, which we do not consider in our paper.

Our cloning model presented in this paper is using a standard, high-level queuing theoretic approach. As described in e.g. [Franks et al., 2008], a computing application can be modeled in greater detail using a layered queuing networks (LQN) approach, where the model is split into smaller components. An interesting possible extension to our work would be to investigate if our proposed techniques also apply to more complex LQN models.

8. Conclusion

This paper presented a theoretical analysis that extends and generalizes known results about request cloning in data centers. We used the concept of synchronized service to denote a certain number of servers that simulta-

neously serve clones of a request. We demonstrated that request cloning in a server system under synchronized service can equivalently be modeled as a G/G/1 server. We showed that *no* assumptions on either inter-arrival or service time distributions are required, and that the G/G/1 model holds for *any* queuing discipline. In this paper, we focused on the PS discipline and show further results for it.

We further extended our theoretical results and discussed the optimal cloning factor. We also analyzed more complex server systems, consisting of multiple clusters. To demonstrate the possible applications of the equivalent G/G/1 modeling, we presented a co-design method that, under homogeneity assumptions, found the optimal cloning factor c_f^{opt} and the server system's corresponding mean response time $\mathbf{E}(T)$ under both random and JSQ load-balancing for clusters with synchronized service. To the best of our knowledge, this paper presents the first provably stable combined load-balancing and cloning strategy for the PS queuing discipline. Further, we relaxed the synchronized service assumption and derived bounds for how practical imperfections, such as arrival and cancellation delays, affect the accuracy of our model. We demonstrated using simulations that removing the synchronized service constraint for the JSQ co-design seems to only marginally reduce the accuracy of the model. We provided an intuitive explanation to this phenomenon, which implies that our theoretical model could be used to design the non-synchronized a-JSQ-d version as well.

Acknowledgments

This work was partially supported by the Wallenberg AI, Auto-nomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Nordforsk Nordic Hub on Industrial IoT (HI2OT), and by the ELLIIT Excellence Center at Lund University.

References

Ananthanarayanan, G., A. Ghodsi, S. Shenker, and I. Stoica (2013). "Effective straggler mitigation: Attack of the clones". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. USENIX Association, Lombard, IL, pp. 185–198.

Ananthanarayanan, G., S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris (2010). "Reining in the outliers in map-reduce clusters using mantri". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. USENIX Association, Vancouver, BC, Canada, pp. 265–278. URL: http://dl.acm.org/citation.cfm?id=1924943.1924962.

Ayesta, U. (2019). "On redundancy-d with cancel-on-start a.k.a join-shortest-work (d)". *ACM SIGMETRICS Performance Evaluation Review* **46**:2, pp. 24–26. DOI: 10.1145/3305218.3305228. URL: https://doi.org/10.1145/3305218.3305228.

Calheiros, R. N., R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya (2011). "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms". *Softw. Pract. Exper.* **41**:1, pp. 23–50.

Dean, J. and L. A. Barroso (2013). "The tail at scale". *Communications of the ACM* **56**:2, p. 74.

Dean, J. and S. Ghemawat (2008). "Mapreduce: simplified data processing on large clusters". *Commun. ACM* **51**:1, pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

Franks, G., T. Al-Omari, M. Woodside, O. Das, and S. Derisavi (2008). "Enhanced modeling and solution of layered queueing networks". *IEEE Transactions on Software Engineering* **35**:2, pp. 148–161.

Gardner, K., M. Harchol-Balter, and A. Scheller-Wolf (2016a). "A better model for job redundancy: decoupling server slowdown and job size". In: *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. DOI: 10.1109/mascots.2016.43. URL: https://doi.org/10.1109/mascots.2016.43.

Gardner, K., S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia (2015). "Reducing latency via redundant requests: Exact analysis". *ACM SIGMETRICS Performance Evaluation Review* **43**:1, pp. 347–360.

Gardner, K., S. Zbarsky, M. Harchol-Balter, and A. Scheller-Wolf (2016b). "The power of d choices for redundancy". *ACM SIGMETRICS Performance Evaluation Review* **44**:1, pp. 409–410. DOI: 10.1145/2964791.2901497. URL: https://doi.org/10.1145/2964791.2901497.

Gupta, V., M. H. Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9-12, pp. 1062–1081. DOI: 10.1016/j.peva.2007.06.012. URL: https://doi.org/10.1016/j.peva.2007.06.012.

Joshi, G. (2018). "Synergy via redundancy: boosting service capacity with adaptive replication". *ACM SIGMETRICS Performance Evaluation Review* **45**:2, pp. 21–28. DOI: 10.1145/3199524.3199530. URL: https://doi.org/10.1145/3199524.3199530.

Joshi, G., Y. Liu, and E. Soljanin (2012). "Coding for fast content download". *CoRR* **abs/1210.3012**. arXiv: 1210.3012. URL: http://arxiv.org/abs/1210.3012.

Joshi, G., E. Soljanin, and G. Wornell (2015). "Efficient replication of queued tasks for latency reduction in cloud systems". In: *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE.

Joshi, G., E. Soljanin, and G. Wornell (2017). "Efficient redundancy techniques for latency reduction in cloud systems". *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* **2**:2, pp. 1–30. DOI: 10.1145/3055281. URL: https://doi.org/10.1145/3055281.

Klein, C., M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez (2014). "Brownout: building more robust cloud applications". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014, pp. 700–711.

Kleinrock, L. (1975). *Queueing Systems. Vol. I: Theory*. Wiley Interscience.

Lee, K., R. Pedarsani, and K. Ramchandran (2017). "On scheduling redundant requests with cancellation overheads". *IEEE/ACM Transactions on Networking* **25**:2, pp. 1279–1290. DOI: 10.1109/tnet.2016.2622248. URL: https://doi.org/10.1109/tnet.2016.2622248.

Modica, G. and L. Poggiolini (2012). *A First Course in Probability and Markov Chains*. John Wiley & Sons, Ltd. DOI: 10.1002/9781118477793. URL: https://doi.org/10.1002/9781118477793.

Qiu, Z., J. F. Pérez, and P. G. Harrison (2016). "Tackling latency via replication in distributed systems". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*. ACM Press.

Shah, N. B., K. Lee, and K. Ramchandran (2014). "The mds queue: analysing the latency performance of erasure codes". In: *2014 IEEE International Symposium on Information Theory*, pp. 861–865. DOI: 10.1109/ISIT.2014.6874955.

Shah, N. B., K. Lee, and K. Ramchandran (2016). "When do redundant requests reduce latency?" *IEEE Transactions on Communications* **64**:2, pp. 715–722. DOI: 10.1109/tcomm.2015.2506161. URL: https://doi.org/10.1109/tcomm.2015.2506161.

Wang, H., J. Li, Z. Shen, and Y. Zhou (2018). "Approximations and bounds for (n, k) fork-join queues: a linear transformation approach". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE. DOI: 10.1109/ccgrid.2018.00069. URL: https://doi.org/10.1109/ccgrid.2018.00069.

Zaharia, M., M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica (2012). "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In: *Presented as part of the 9th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI 12). USENIX, San Jose, CA, pp. 15–28. URL: `https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia`.

Zaharia, M., A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica (2008). "Improving mapreduce performance in heterogeneous environments". In: *8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California, pp. 29–42.

Paper V

Towards Performance Modeling of Speculative Execution for Cloud Applications

Tommi Nylander **Johan Ruuskanen** **Karl-Erik Årzén**
Martina Maggio

Abstract

Interesting approaches to counteract performance variability within cloud datacenters include sending multiple request clones, either immediately or after a specified waiting time. In this paper we present a performance model of cloud applications that utilize the latter concept, known as speculative execution. We study the popular Join-Shortest-Queue load-balancing strategy under the processor sharing queuing discipline. Utilizing the *near-synchronized service* property of this setting, we model speculative execution using a simplified synchronized service model. Our model is approximate, but accurate enough to be useful even for high utilization scenarios. Furthermore, the model is valid for any, possibly empirical, inter-arrival and service time distributions. We present preliminary simulation results, showing the promise of our proposed model.

© 2020 ACM/SPEC. Originally published in International Conference on Performance Engineering Companion, Edmonton, July 2020. Reprinted with permission. The article has been reformatted to fit the current layout.

1. Introduction

Speculative execution is a popular method employed by cloud providers as a tool for increasing predictability of the execution time of jobs [Dean and Barroso, 2013; Ren et al., 2015]. Redundancy is introduced by launching copies of tasks that have been running for an unusually long time. The general idea is that the unpredictability of task execution times due to effects such as resource contention or network queues, can be mitigated by identifying slow running instances and launching copies that will hopefully complete before the original.

A closely related topic which has recently received increased attention from researchers is cloning. As explained by Ganesh et al. [Ananthanarayanan et al., 2013], cloning can be seen as a special case of speculative execution with no speculation time, i.e. all clones are sent immediately. We refer to [Nylander et al., 2020] for more related work on this topic.

Much of the research on speculative execution have been done considering the case of straggler mitigation in distributed computing using big data frameworks such as MapReduce [Dean and Ghemawat, 2008]. Here a job is split into several tasks, and is not considered completed until all, or a subset, of tasks have been completed. The ultimate response time of a job is thus highly sensitive to slow running tasks. Modeling and analysis of such systems often either assume that each server can only take a single job at a time [Xu and Lau, 2017], or that the introduction of redundancy does not affect the service times of other jobs. An exception for the latter is the recent contribution of Aktas et al. [Aktaş and Soljanin, 2019], which shortly considers the effect of redundancy on the response time distribution of tasks.

In this work we instead consider the case of replicated cloud applications subject to independent user requests, and seek to model the behaviour of such systems under speculative execution from a queuing model perspective. Our approach is dependent on two key concepts from [Nylander et al., 2020], summarized in the following paragraph.

Near-Synchronized Service. The concept of *synchronized service* was introduced in [Nylander et al., 2020] to simplify modeling of request cloning. Its full definition is given in [Nylander et al., 2020], but in short, cloning under synchronized service implies that the clone that completes first is the one that receives the shortest service time. For the processor sharing (PS) queuing discipline, synchronized service implies that all n request clones $r^c_{1:n}$ of an original request r^o experience *identical* processor shares across all n servers. As synchronized service is very difficult to achieve in practice, the concept of *near-synchronized* service was further introduced in [Nylander et al., 2020] to model scenarios that include imperfections such as arrival and cancellation delays. Additionally, it was shown empirically that the widely used load-balancing strategy Join-Shortest-Queue (JSQ) provides near-synchronized

service for all request clones $r^c_{1:n}$, when using PS as queuing discipline. This property is very interesting as it allows for accurate approximate performance modeling of JSQ cloud applications subject to cloning, by using a simplified synchronized service model.

Contributions. Using the near-synchronization property of JSQ under PS, we (i) derive a novel performance model for replicated cloud applications subject to speculative execution; (ii) assuming Poisson arrivals, use existing results from queuing theory to obtain an approximate yet accurate expression for the average response time; and (iii) empirically demonstrate the potential of our model through simulations.

2. Model

We consider performance modeling of a cloud application replicated over m homogeneous servers, modeled using the PS queuing discipline with service rate μ. User requests r_o with rate λ enter at the load balancer, leading to a system utilization $\rho = \lambda/(m\mu)$. The requests are dispatched to the servers using the JSQ strategy, that always chooses the least occupied server. We do not assume any specific distributions, however, for simplicity we require the service times to be independent and identically distributed (i.i.d.) across all m servers. When a specified amount of *service time* has been processed for an original request r_o, a speculative clone r^i_s is dispatched to a unique server, again using JSQ. This server system under JSQ and PS was shown in [Nylander et al., 2020] to provide near-synchronized service, while our modeling approach is performed assuming synchronized service. Our derived performance metrics, including utilization ρ and average response time T, are thus approximate.

Define s_i as the service time when the speculative clone r^i_s is dispatched to the server system and $\mathcal{S}_n = \{s_1, s_2, \ldots, s_n\}$ as the ordered set of the service times of all speculative cloning instances with $s_{i-1} \leq s_i$. Denote by $F(x)$ the cumulative distribution function (CDF), and F^0_r as the original service time CDF. Using Theorem 2 in [Nylander et al., 2020], the following iterative formula (1) can be used to determine the resulting service time CDF F^n_r for the speculation scenario $\mathcal{S}_n$:

$$F^i_r(x) = \begin{cases} F^0_r(x), & x \leq s_1 \\ F^0_r(s_1) + (1 - F^0_r(s_1)) \cdot F^1_c(x), & s_1 < x \leq s_2 \\ \vdots & \\ F^{i-1}_r(s_i) + (1 - F^{i-1}_r(s_i)) \cdot F^i_c(x), & s_i < x \end{cases} \quad (1)$$

with the intermediate CDF $F_c^i(x)$ determined as

$$F_a^i(x) = F_r^{i-1}(x|s_i < x) \tag{2}$$
$$F_b^i(x) = F_r^0(x - s_i) \tag{3}$$
$$F_c^i(x) = 1 - (1 - F_a^i(x)) \cdot (1 - F_b^i(x)). \tag{4}$$

The algorithm is visualized in Figure 1 for an example scenario $\mathcal{S}_2$.

From (1), we can calculate the new average service rate $\mu(\mathcal{S}_n)$ for a scenario using n speculative clones as

$$\mu(\mathcal{S}_n) = \left(\int_0^\infty (1 - F_r^n(x))dx\right)^{-1}. \tag{5}$$

We define the *service factor* $f_\mu(\mathcal{S}_n)$ as the normalized increase of $\mu(\mathcal{S}_n)$ compared to the original $\mu(\mathcal{S}_0) = \mu$:

$$f_\mu(\mathcal{S}_n) = \frac{\mu(\mathcal{S}_n)}{\mu}. \tag{6}$$

To model the changes to the server system load, we need to consider the amount of speculative clones sent for each original request r_o and the time they spend in the system. We define the *speculation factor* f_p^i for a speculative clone at time s_i as the probability $f_p^i = 1 - F_r^i(s_i)$ that the clone is sent. Furthermore, we define the *sojourn factor* f_s^i for a speculative clone sent at time s_i as its time spent in the system compared to the original request r_o

$$f_s^i = \frac{\int_{s_i}^\infty (1 - F_r^n(x|s_i < x))dx}{\int_0^\infty (1 - F_r^n(x))dx}. \tag{7}$$

Now we define the *arrival factor* $f_\lambda(\mathcal{S}_n)$ for the total contributions to system load from all n speculative clones as

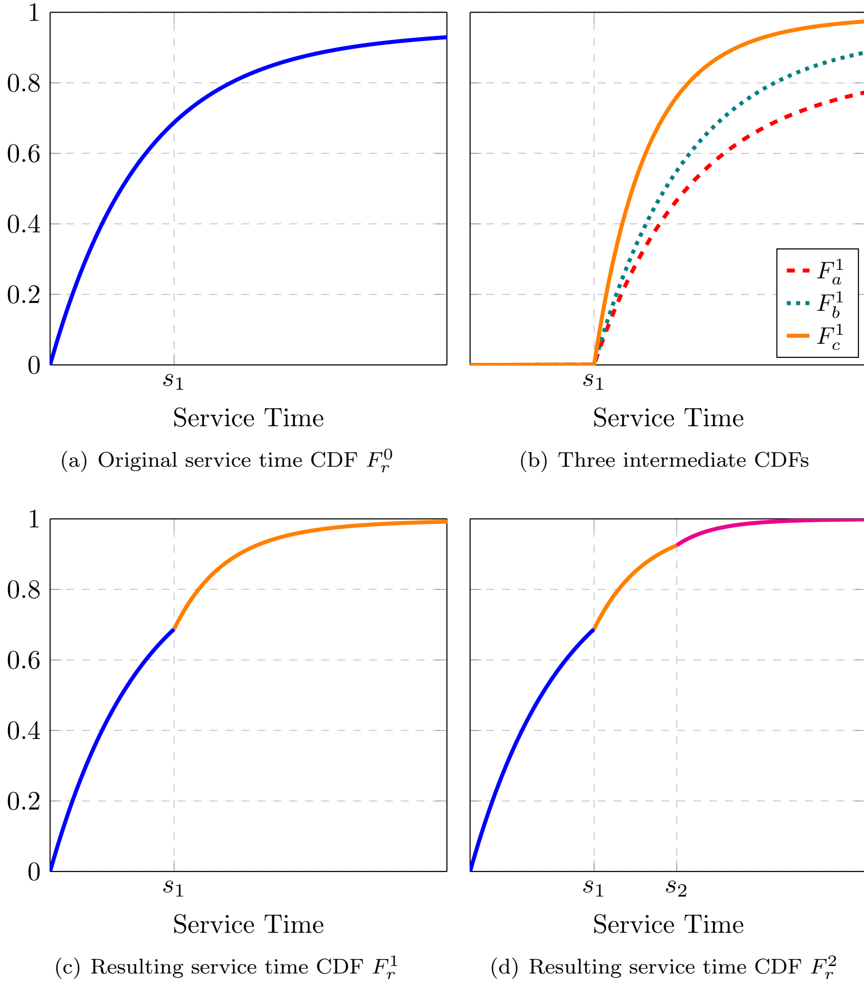
$$f_\lambda(\mathcal{S}_n) = 1 + \sum_{i=1}^{n} f_p^i \cdot f_s^i. \tag{8}$$

Finally, the *load factor* $f_\rho(\mathcal{S}_n)$ can then be defined as

$$f_\rho(\mathcal{S}_n) = \frac{f_\lambda(\mathcal{S}_n)}{f_\mu(\mathcal{S}_n)}. \tag{9}$$

$f_\rho(\mathcal{S}_n) > 1$ thus means that speculative cloning under scenario $\mathcal{S}_n$ results in an increase of the original system load $\rho(\mathcal{S}_0) = \rho$, whereas $f_\rho(\mathcal{S}_n) < 1$ represents a decrease. Also, we can define the modeled utilization of scenario $\mathcal{S}_n$ as

$$\rho(\mathcal{S}_n) = f_\rho(\mathcal{S}_n) \cdot \rho. \tag{10}$$



(a) Original service time CDF F_r^0

(b) Three intermediate CDFs

(c) Resulting service time CDF F_r^1

(d) Resulting service time CDF F_r^2

Figure 1. Speculative cloning for an example scenario $\mathcal{S}_2 = \{s_1, s_2\}$. From F_r^0 at s_1, three intermediate CDFs are formed using equations (2)-(4). Then F_c^1 is added to F_r^0 at s_1 according to Eq. (1) to form F_r^1. The procedure is then repeated at s_2 to form F_r^2.

Equation (10) is very useful as it allows us to determine stability for the scenario $\mathcal{S}_n$ by studying if $\rho(\mathcal{S}_n) < 1$. Note that the arrival factor $f_\lambda(\mathcal{S}_n) > 1$ does not imply an increase in the arrival rate of original requests r_o, i.e. $\lambda(\mathcal{S}_n) = \lambda(\mathcal{S}_0) = \lambda$ for all n. Instead, it represents the contributions to the system load from all speculative clones. We model this as a decrease in the number of available servers $m(\mathcal{S}_0) = m$ as

$$m(\mathcal{S}_n) = \frac{m}{f_\lambda(\mathcal{S}_n)}. \tag{11}$$

As a result, $m(\mathcal{S}_n) \in \mathbf{R}^+$ is defined as a positive real number. Note that for non-speculative cloning (with all clones sent at $s_i = 0$), $f_\lambda(\mathcal{S}_n)$ and $m(\mathcal{S}_n)$ assume integer values. The clone-to-clusters model in [Nylander et al., 2020], which divides n servers into m clusters thus fits as a special case in our speculative execution model.

To be able to get explicit response time measures, we need to assume Poisson arrival rates for $\lambda(\mathcal{S}_n)$. This allows us to utilize the very accurate (within 2-3%) approximate response time model for JSQ under PS from [Gupta et al., 2007]. It provides average response times $T(\mathcal{S}_n)$ from the inputs (i) arrival rate $\lambda(\mathcal{S}_n)$; (ii) service rate $\mu(\mathcal{S}_n)$; and (iii) number of servers $m(\mathcal{S}_n)$.

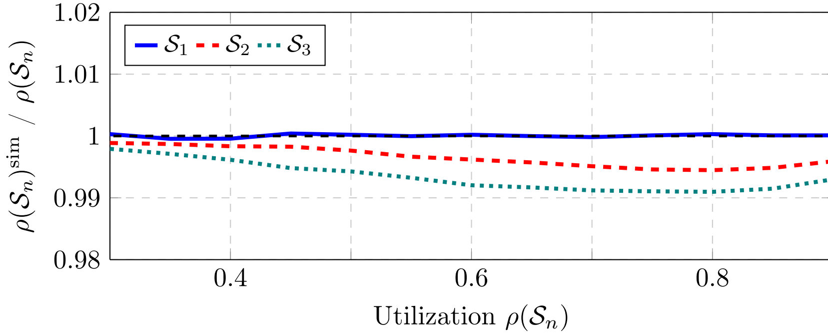
Using a simplified synchronized service approach, we are thus able to approximately model utilization, stability and average response times for a replicated cloud application under a speculation scenario $\mathcal{S}_n$, assuming a JSQ+PS setup. The model accuracy is a potential issue that is examined in the next section. Another drawback with our approach is that it might be complicated to implement triggering of speculative clones at processed service times s_i as these can be cumbersome to keep track of in a real system.

We evaluate our model using a discrete-event simulator, based on the `cloning-simulator` from [Nylander et al., 2020] but extended with support for speculative execution. We use Poisson arrivals and our service times are distributed as Pareto (Type 1, shape=2.1, scale=0.5). We simulate using $m = 10$ servers under system loads $\rho(\mathcal{S}_n)$ from 0.3 to 0.9 and consider three different speculation scenarios: (i) $\mathcal{S}_1 = \{1.5\}$; (ii) $\mathcal{S}_2 = \{0.7, 1.0\}$; and (iii) $\mathcal{S}_3 = \{0.3, 0.6, 0.9\}$ (all units in seconds).

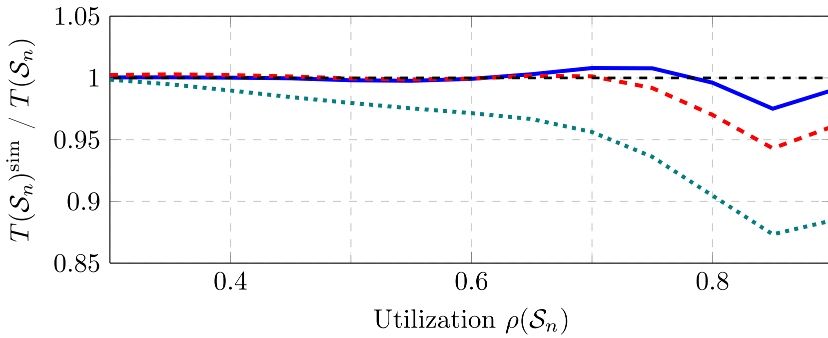
3. Evaluation

Figure 2 shows our preliminary results. In Figure 2(a), the simulated system utilization $\rho(\mathcal{S}_n)^{\text{sim}}$ is normalized against our modeled $\rho(\mathcal{S}_n)$. The results are very close to 1 for all scenarios and loads, which points towards that our model is very accurate at predicting utilization and stability.

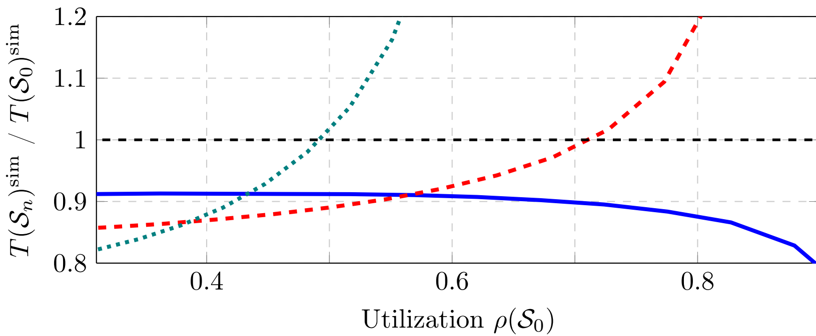
Figure 2(b) shows the results of the simulated average response times $T(\mathcal{S}_n)^{\text{sim}}$ normalized against our modeled $T(\mathcal{S}_n)$. As can be seen, the accu-



(a) Simulated vs modeled $\rho(\mathcal{S}_n)$.



(b) Simulated vs modeled $T(\mathcal{S}_n)$.



(c) Speculation scenarios $\mathcal{S}_n$ vs $\mathcal{S}_0$.

Figure 2. Simulation results using 20 repeated runs with 10^6 requests. Confidence intervals are tight and left out for readability.

racy of our model is very high for low to medium loads for all three speculation scenarios. However, for higher loads our model accuracy is worse (but still reasonable) for the more complicated scenarios. A probable explanation is that the service is further away from synchronization here. The final Figure 2(c) shows the results of the simulated average response times $T(\mathcal{S}_n)^{\mathrm{sim}}$ for the speculation scenarios normalized against $T(\mathcal{S}_0)^{\mathrm{sim}}$, where no speculation is present. A value below 1 indicates that the speculation scenarios are beneficial, and as can be seen all three scenarios perform well for low loads. Scenario $\mathcal{S}_1$ distinguishes itself from the the other two by actually outperforming the no speculation case at all system loads. The reason is that its load factor $f_\rho(\mathcal{S}_1)$ is below 1, i.e. it always *decreases* the system load. This is very interesting as it can be shown, using techniques from [Nylander et al., 2020], that standard cloning (all $s_i = 0$) under this particular Pareto distribution is only beneficial for low loads. Speculative execution thus has the potential to be more useful than cloning under high loads.

4. Conclusion

We have presented a novel model of a replicated cloud application subject to speculative execution, that looks promising in our preliminary evaluation. We plan to expand our evaluation to be more general, and to use our model to find optimal speculation configurations $\mathcal{S}_n^\star$, providing the shortest response times. A possible approach could be to search for the configurations that minimize the system utilization, in order to provide performance enhancements even for server systems under high load.

Acknowledgments

This work was partially supported by the Wallenberg AI, Auto-nomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, by the Nordforsk Nordic Hub on Industrial IoT (HI2OT), and by the ELLIIT Excellence Center at Lund University.

References

Aktaş, M. F. and E. Soljanin (2019). "Straggler mitigation at scale". *IEEE/ACM Transactions on Networking* **27**:6, pp. 2266–2279. ISSN: 1558-2566.

Ananthanarayanan, G., A. Ghodsi, S. Shenker, and I. Stoica (2013). "Effective straggler mitigation: Attack of the clones". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. USENIX Association, Lombard, IL, pp. 185–198.

Dean, J. and L. A. Barroso (2013). "The tail at scale". *Communications of the ACM* **56**:2, p. 74.

Dean, J. and S. Ghemawat (2008). "Mapreduce: simplified data processing on large clusters". *Commun. ACM* **51**:1, pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

Gupta, V., M. H. Balter, K. Sigman, and W. Whitt (2007). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9-12, pp. 1062–1081. DOI: 10.1016/j.peva.2007.06.012. URL: https://doi.org/10.1016/j.peva.2007.06.012.

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020). "Modeling of request cloning in cloud server systems using processor sharing". In: *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada*.

Ren, X., G. Ananthanarayanan, A. Wierman, and M. Yu (2015). "Hopper: decentralized speculation-aware cluster scheduling at scale". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication - SIGCOMM '15*.

Xu, H. and W. C. Lau (2017). "Optimization for speculative execution in big data processing clusters". *IEEE Transactions on Parallel and Distributed Systems* **28**:2, pp. 530–545. ISSN: 2161-9883.