LUND UNIVERSITY

Dynamical Modeling of Cloud Applications for Runtime Performance Management

Ruuskanen, Johan

2022

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

Link to publication

Citation for published version (APA):
Ruuskanen, J. (2022). *Dynamical Modeling of Cloud Applications for Runtime Performance Management*. [Doctoral Thesis (monograph), Department of Automatic Control]. Department of Automatic Control, Lund Institute of Technology, 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

Download date: 08. Nov. 2022

Dynamical Modeling of Cloud Applications for Runtime Performance Management

Johan Ruuskanen

LUND
UNIVERSITY

Department of Automatic Control

PhD Thesis TFRT-1137
ISBN 978-91-8039-393-5 (print)
ISBN 978-91-8039-394-2 (web)
ISSN 0280-5316

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

© 2022 by Johan Ruuskanen. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2022

Abstract

Cloud computing has quickly grown to become an essential component in many modern-day software applications. It allows consumers, such as a provider of some web service, to quickly and on demand obtain the necessary computational resources to run their applications. It is desirable for these service providers to keep the running cost of their cloud application low while adhering to various performance constraints. This is made difficult due to the dynamics imposed by, e.g., resource contentions or changing arrival rate of users, and the fact that there exist multiple ways of influencing the performance of a running cloud application. To facilitate decision making in this environment, performance models can be introduced that relate the workload and different actions to important performance metrics.

In this thesis, such performance models of cloud applications are studied. In particular, we focus on modeling using queueing theory and on the fluid model for approximating the often intractable dynamics of the queue lengths. First, existing results on how the fluid model can be obtained from the mean-field approximation of a closed queueing network are simplified and extended to allow for mixed networks. The queues are allowed to follow the processor sharing or delay disciplines, and can have multiple classes with phase-type service times. An improvement to this fluid model is then presented to increase accuracy when the *system size*, i.e., number of servers, initial population, and arrival rate, is small. Furthermore, a closed-form approximation of the response time CDF is presented. The methods are tested in a series of simulation experiments and shown to be accurate.

This mean-field fluid model is then used to derive a general fluid model for microservices with interservice delays. The model is shown to be completely extractable at runtime in a distributed fashion. It is further evaluated on a simple microservice application and found to accurately predict important performance metrics in most cases. Furthermore, a method is devised to reduce the cost of a running application by tuning load balancing parameters between replicas. The method is built on gradient stepping by applying automatic differentiation to the fluid model. This allows for arbitrarily defined

cost functions and constraints, most notably including different response time percentiles. The method is tested on a simple application distributed over multiple computing clusters and is shown to reduce costs while adhering to percentile constraints.

Finally, modeling of request cloning is studied using the novel concept of synchronized service. This allows certain forms of cloning over servers, each modeled with a single queue, to be equivalently expressed as one single queue. The concept is very general regarding the involved queueing discipline and distributions, but instead introduces new, less realistic assumptions. How the equivalent queue model is affected by relaxing these assumptions is studied considering the processor sharing discipline, and an extension to enable modeling of speculative execution is made. In a simulation campaign, it is shown that these relaxations only has a minor effect in certain cases.

Acknowledgments

I would like to start by thanking the Department of Automatic Control for giving me this opportunity and my supervisors Anton Cervin and Karl-Erik Årzén for guiding me during my time as a PhD student. A special thanks to my main supervisor Anton, who has always been supportive and available for discussing ideas, proof-reading and writing papers.

Furthermore, I would like to extend my gratitude to everyone at the department for making these past five years an interesting and joyful experience, and to all the new friends that I have made. A special thanks to Tommi Berner and Albin Heimerson for the compelling discussions and collaboration on cloud related topics, Marcus Thelander Andrén for the spontaneous talks on event-based control and estimation, Anders Robertsson for our teaching visit to Beihang University, and Alexandre Martins and Martin Heyden for introducing me to the department. I am moreover very grateful for all the good times I have had at the Friday study circle and other important extracurricular activities and to the people who made them happen. Especially to Claudio Mandrioli, Marcus Greiff, and Nils Vreman for the fun trips to Montreal and Vietnam, but also for the planned trip to Iceland that was canceled due to the sudden bankruptcy of the airline two weeks prior.

I would like to thank Wallenberg AI, Autonomous Systems and Software Program (WASP) for funding my position and for all the fun and informative study trips, courses, and networking events. I extend my gratitude to Ericsson Research for granting access to their private cloud Xerces, allowing me to tinker with all manners of cloud-related software, which played an important role in my research. I am also grateful to Erik Elmroth for organizing the biannual cloud control workshops and to Lars Larsson and Johan Eker for sparking my curiosity for cloud computing, which ultimately inspired me to alter my research topic. Furthermore, I would like to thank Nils Vreman, Johan Eker, Giacomo Como, Albin Heimerson, Max Nyberg Carlsson, and Martina Maggio for their invaluable but taxing job of proof-reading this thesis.

Thanks to all the friends I met in the student orchestra Alte Kamereren for the fun we have had during these years and the 60 days in total I now apparently have spent touring the biergartens of Germany. To my old friends Carl, Erik, Martin, Niklas, Patrik, Per, Sakarias, and Torkel, thank you for all the hang-outs and happenings that have helped me keep my mind off work. To my parents, Maria and Seppo, and to my sister Charlotta, thank you for your unwavering support and encouragement.

Last but not least, to my better half Micki, thank you for putting up with all my quirks and for bringing joy and purpose to my life.

Financial Support

This thesis was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Johan Ruuskanen is a member of the ELLIIT Excellence Center at Lund University.

Contents

Nomenclature

Notation	Description
$\mathbb{R}_{(+)}$, $\mathbb{Z}_{(+)}$	Set of (nonnegative) real numbers and integers.
$\lvert\cdot\rvert$	Number of elements in a set.
$(a_1, a_2, \dots)$ $[a_1, a_2, \dots]$	Two ways of writing a tuple of two or more elements.
$[a, b],\ a, b \in \mathbb{R}$	Closed interval of real numbers between a, b.
$a : b,\ a, b \in \mathbb{Z}$	Closed interval of integers between a, b.
$(a, b) \in (\mathcal{A}, \mathcal{B}_a)$	Set nesting, implies $b \in \mathcal{B}_a$ where $a \in \mathcal{A}$.
$\mathbb{1}\,(e)$	Function yielding 1 if expression e is true, else 0.
$\mathbb{R}^{a\times b}$	Matrix of real numbers with a rows and b columns.
A^T	Transpose of matrix A.
$A \odot B$	Elementwise product between two matrices A, B.
$Y \sim G$	Stochastic variable Y with distribution G.
$\mathbb{E}\,[Y]$	Expected value of stochastic variable Y.
$\Phi(y)$	Cumulative distribution function.
$\mathcal{Q}$	Set of queues in a queueing network.
$\mathcal{C}_q$	Set of classes for queue q.
$\mathcal{S}_{q,c}$	Set of phase states for $(q, c) \in (\mathcal{Q}, \mathcal{C}_q)$.
Γ	Set of chains in a queueing network.
$Q(a)$	Function mapping phase state or class to its queue.
k_q	Number of servers in queue q.
t^a_r, t^s_r, t^d_r	Arrival, service and departure time of request r.
$\mathcal{H}$	Tracing data.
λ, μ	Arrival and service rates.
ρ, τ	Utilization and throughput.
X, T	Queue length and response time.
φ	Response time percentile, often 95th.
$\boldsymbol{P}$	Routing probability matrix, between classes.
$\{\Psi, \psi, \zeta\}_{q,c}$	PH distribution matrices in $(q, c) \in (\mathcal{Q}, \mathcal{C}_q)$.
$\boldsymbol{\Psi}, \boldsymbol{B}, \boldsymbol{A}$	Block diagonals of the stacked PH matrices.
$\boldsymbol{W}$	State transition rate matrix, formed as $\boldsymbol{\Psi} + \boldsymbol{BPA}^T$.
$\theta(\boldsymbol{X})$	State transition rate function.
$g(x)$	Processor share function.
η	Smoothing parameter.

Abbreviation	**Description**
BCMP	Theorem for identifying product-form networks.
CDF	Cumulative distribution function.
CoC, CoS	Cancel-on-Complete/Start cloning policy.
CoV	Coefficient of variation.
CTMC	Continuous-time Markov chain.
FCFS	First-come-first-served queueing discipline.
FedApp	Federated application sandbox.
IaaS	Infrastructure-as-a-service.
IID	Independent and identically distributed.
INF	Delay, or infinite, queueing discipline.
JSQ	Join-shortest-queue load balancing policy.
LQN	Layered queueing network.
LQR	Linear quadratic regulator.
M/G/1-FCFS	Kendall's notation of a queue, here with - Markovian interarrival time distribution, - General service time distribution, - Single server and FCFS discipline.
MPC	Model predictive control.
MVA	Mean value analysis.
ODE	Ordinary differential equation.
p95	95th percentile.
PaaS	Platform-as-a-service.
PDF	Probability density function.
PH	Phase-type distribution.
PMF	Probability mass function.
PS	Processor sharing queueing discipline.
PSFFA	Pointwise stationary fluid flow approximation.
QoS	Quality of service.
RE, AE	Relative, or absolute, error.
SaaS	Software-as-a-Service.
SLA	Service level agreement.
SLO	Service level objective.
SOA	Service oriented architecture.
SV	Stochastic variable.
VM	Virtual Machine.

1

Introduction

It is hard to imagine everyday life without all of our countless smartphone apps and web services that we have come to rely so heavily upon. At the tip of our fingers, as long as we have a stable internet connection, we are conveniently provided with direct access to a wide range of services in the form of search engines, social media, video streaming, and banking services, to name a few. This has had a profound impact on society, recently made evident in the ongoing Covid pandemic with the spread of information and disinformation on social media platforms, and as a key enabler of remote working through video conference applications such as Zoom or Microsoft Teams. Furthermore, in certain cases, these services have come to supplant more traditional means. A clear example is searching for information that nowadays is mostly performed online, and even has its own verb: *to google*[1].

Running many of these services directly on local devices, be it your laptop or smartphone, would be unfeasible due to a number of reasons. Depending on the application, there might be, for example, too little computation power or storage to spare at the local device or simply a dependence on other users. The service provider can instead put some parts of the application on a remote server, which the local device then can communicate to via, e.g., an app or a web browser. This server can then be fitted with enough hardware to adequately serve all its users. However, in order to ensure some sort of *Quality of Service* (QoS) for the users, there needs to be enough hardware to handle worst-case scenarios in day-to-day user traffic. Furthermore, the cost to setup and maintain a computing infrastructure is a hindrance, particularly for startups and small companies that might lack the upfront funds required. Finally, if demand increases, it will take time to scale the service to accommodate new users, as this requires purchasing and installing new hardware components. Hence, it is seldom practical for service providers to have their own dedicated physical hardware for their services.

[1] https://www.dictionary.com/browse/google

Instead, service providers can leverage *cloud computing* to rent *virtual* resources in data centers hosted by some cloud provider, and thus obtain (and pay for) the necessary compute power, network, and storage only when it is needed. This essentially enables the service provider to treat the hardware as an *on-demand utility*. However, how to leverage these on-demand resources is not trivial, as the number of users and the type of user request typically change over time. Furthermore, virtualized hardware has varying performance, and multiple users sharing the same physical hardware can give rise to resource contentions. This leads to unpredictable application performance and the need to dynamically make decisions in order to ensure some QoS while keeping costs low. The type of action to take is also a challenge. Depending on the specific situation, it might be appropriate to perform some resource scaling, service migration, or re-tuning of load balancing weights between multiple instances of the application. Ideally, to ensure speed and cost effectiveness, these actions should be taken automatically with minimal interaction from human operators. To automatically make well-informed decisions, a good *performance model* of the application is needed to infer how a particular action would affect the QoS.

It is the purpose of this thesis to study such performance models of cloud applications, in order to enable better automatic decision making at runtime. In particular, a certain type of dynamical performance model known as the *fluid model* will be studied, and how it can be used to decide cost minimizing strategies for load balancing in a running cloud application. In addition, the thesis studies request redundancy as a way of reducing response times.

1.1 Cloud Computing

In essence, cloud computing is a business model, as quickly explained in Figure 1.1. By leveraging the economy of scale[2] and the placement of cheap electricity, a cloud provider can operate its data centers at a relatively low cost. This allows cloud providers to sell computational power and storage at a cheaper rate than what it would cost service providers to own and operate their needed resources. To put things in perspective, a large scale data center with tens of thousands of servers can operate around 5 times cheaper, even compared to a medium sized of hundreds to thousands of servers [Armbrust et al., 2009]. Furthermore, an essential key feature of cloud computing is the ability for consumers renting resources to quickly obtain and release resources when the need arise. Thus there is no need to pay upfront for the hardware nor the resources not currently used.

[2] Cost for hardware, bandwidth, software development and operations does not increase linearly with the amount of servers.

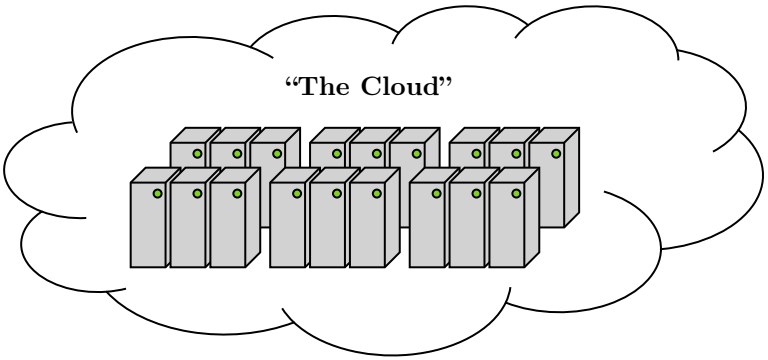


Figure 1.1 *The Cloud is just someone else's computer* is a popular saying. Although there is a grain of truth in this statement, that someone can operate those computers at a low cost and allow consumers to utilize parts of the pooled computational resources on demand.

Therefore, when comparing operational costs, renting hardware on demand is the preferable choice. This is further made evident in the widespread adoption and growth of cloud computing. For example, in 2021 an average of 41% of all enterprises in the European Union utilized cloud computing in some fashion, an increase of 5% from 2020. In certain member states, such as Sweden, the percentage is as high as 75%[3]. Furthermore, in 2021 global cloud spending amounted to a total of $707 billion, and is forecast to grow to a staggering total of $1.3 trillion by 2025 according to IDC[4].

The concept of cloud computing is very broad and goes beyond service providers renting virtualized hardware. According to the NIST definition [Mell, Grance, et al., 2011], cloud computing terminology can basically be applied to any compute infrastructure that provides resources to consumers and that (i) pools its own computing resources to serve multiple consumers simultaneously, (ii) is accessible via a network, and (iii) measures the provided resources for both internal management as well as transparency on usage for the provider and consumer. Further, the compute infrastructure needs to enable (iv) self-provisioning of resources without the need to interact with a human operator, and (v) quick response in provision and release of resource demands for its consumers.

Deployment models. Cloud infrastructures can be categorized into *deployment models* depending on the relationship between the cloud provider and its customers. The following three models are common:

[3] https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises

[4] https://www.idc.com/getdoc.jsp?containerId=prUS48208321

- *Public cloud* – Operated by, e.g., a business for the general public.
- *Community cloud* – Operated by a community of organizations, sharing some common interest, for its members.
- *Private cloud* – Operated by a single organization for internal use.

A fourth common category is the *hybrid cloud*, which is a combination of distinct deployment models linked by some software for easy use.

Service models. How cloud computing can be utilized can also be split into different categories, known as *service models*. Different service models can have different types of resources it provides, not necessarily just virtualized hardware, but also, e.g., access to specific applications. Furthermore, service models provide different levels of autonomy for the consumer. The following three are standard.

- *Infrastructure-as-a-service* (IaaS) – The consumer can directly obtain different computational resources, such as processing, storage, and network, often in the form of *virtual machines* (VMs). The management of these resources to—e.g., deploy an application—is left to the consumer.
- *Platform-as-a-service* (PaaS) – The consumer can obtain an application-hosting environment. Within the bounds of this environment, the consumer can then deploy and manage an application, but does not have to manage the underlying computational resources which is left to the cloud provider.
- *Software-as-a-service* (SaaS) – The consumer can gain access to a specific software application, deployed on some infrastructure. The application, its deployment, and the necessary resources are managed in its entirety by the provider.

Conceptually, our aforementioned service providers can be seen as providing a SaaS for their users, while some cloud provider provides the IaaS from which the providers rents the virtualized hardware.

Cloud Applications

No matter the deployment or service model used, a cloud application is some software running in the cloud that users can interact with by sending *requests* over a communication network. Upon arrival, the request is processed in the application, and on completion a response is often returned to the sending user. Multiple requests can be processed simultaneously, and how the resources are shared between them depends both on the implementation and application type. For some cloud applications, it might be possible to process

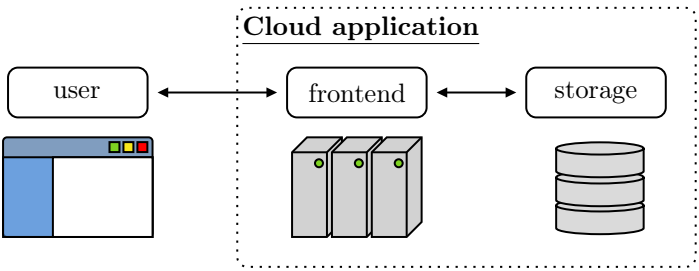


Figure 1.2 Modern cloud applications are often comprised of services that can be deployed separately. In this toy example, an application of a frontend and a storage part has been split into two different services, deployed on two separate types of VMs to better facilitate their needs.

each user request simultaneously by, for example, assigning the processing of each request to a concurrently running thread. For others, access to specific hardware components, e.g., a disc, might limit requests to be processed one-by-one in some order depending on attributes such as user priority or time of request arrival.

To run the application in the cloud, it must be *deployed* in some manner. Arguably, the most straightforward way is simply to pack the necessary software into a single *monolithic* unit and run it on a single virtual machine acquired from an IaaS provider. In this way, it becomes very simple to migrate existing applications from on-premise servers to the cloud. However, such a simple application deployment has its downsides. Monolithic deployments lack modularity, making it difficult for teams to develop and maintain larger applications, to roll out smaller updates and bugfixes, and to identify sources of errors. This also implies that monolithic deployments need to be managed as a single unit. If there are any constrains on parts of the application, such as the requirement for expensive access to specific hardware, the entire monolith will have this constraint. Furthermore, assigning and releasing resources to cope with changing demands must be done for the entire monolith, not just the parts that constitute bottlenecks.

Instead, modern applications are often comprised of smaller parts, or *services*, each performing a small set of specific and related tasks. Using common communication protocols, the services can cooperate to jointly deliver the functionalities of the application. As an example, consider a simple storage application containing a user-facing frontend part and a storage part. It might be preferable to split the application into a frontend and a storage service, and deploy the frontend on VMs with high processing capabilities and even closer to their end users in a *multi-cloud* deployment for low latency. The storage deployment can instead be reserved for VMs with access to more disc. See Figure 1.2 for an illustration.

The popularity of the cloud has further led to a plethora of emerging technologies, solving old challenges and enabling new opportunities for cloud computing [Buyya et al., 2018]. Multi-cloud strategies including small local datacenters in *fog/edge computing*, increased focus on decomposition of applications into disjoint services under the *microservice* architecture, and the use of lightweight *containers* instead of more cumbersome VMs for isolating workloads are just some examples that are seeing increased adoption. For example, Flexera[5] reported that almost all of their surveyed enterprises in 2021 used multi-cloud strategies in some manner, and more than half some type of container technology.

Fog/edge computing. Due to its early stage, the fog/edge computing concept encompasses a multitude of different definitions. But essentially, from a cloud application point of view, it is all about moving the application, or parts of it, to smaller local datacenters or devices closer to the end user. Thus, the concept has strong connections to multi-cloud deployments. Fog/edge computing seeks to address applications with security critical or latency sensitive components, or that produce massive amounts of data, hindering a deployment on a faraway data center. Small local clouds, potentially owned by the users for security concerns, could instead host the constrained components or perform, e.g., data preprocessing to reduce the data transmitted en masse. Such applications are expected to increase in the near future, in part due to the projected growth of connected devices in the so-called *Internet-of-Things* (IoT) [Yousefpour et al., 2019].

Microservice architecture. The modern microservice architecture emphasizes service modularity when decomposing applications into smaller parts. The key difference from other related architectures, such as the mainstream *service oriented architecture* (SOA), is the focus on that each service should be lightweight, have only one (or very few) task(s) it should perform, and that services should be independent of each other, both in deployment and management. A small size makes each service more comprehensible to developers, which simplifies development and maintenance. Additionally, their mutual independence makes it easy to deploy, run updates, and assign/release resources to individual services. In all, these benefits make microservices very suitable for advanced applications deployed in the cloud, and a shift to this architecture is currently underway [Cerny et al., 2018]. This is further evident by its inclusion in the Cloud Native Computing Foundation charter[6], an organization with heavy industrial backing and part of the nonprofit Linux Foundation, with the goal of making technologies for building and running cloud tailored applications ubiquitous.

[5] https://resources.flexera.com/web/pdf/report-cm-state-of-the-cloud-2021.pdf?elqTrackId=28d62429a6ec40d0bb8e92159e68d63a&elqaid=6545&elqat=2

[6] https://github.com/cncf/foundation/blob/main/charter.md

1.2 Application Performance

Running an application in the cloud is one thing, but running it in a way that is satisfactory to its users is another. Poor performance can be disruptive for operations relying on the application, or even outright dangerous if the application is mission critical. Furthermore, dissatisfied users can abandon the provider in favor of competitors or other technologies.

However, what constitutes satisfactory performance depends on the type of application and the need of its users. To specify this, it is normal to set up a *service level agreement* (SLA)[7] between the provider and its users. The SLA provides details about, and requirements on, e.g., what service is provided, the responsibilities of the users and provider, and the quality of the performance. An SLA can take the form of an explicit agreement to a paying user or be implicitly based on user expectations. The consequences of not meeting the requirements are also specified in the SLA; for explicit agreements this could, e.g., be an agreed upon cost reduction for the user.

Quantifying performance. To quantify the SLA requirements, it is standard to create *service level objectives* (SLOs) that define limits on measurable QoS or *performance* metrics that should be fulfilled. Although application specific, common metrics on which to base an SLO are [Beyer et al., 2016]

- *Availability* – The fraction of time the application is up and running.
- *Response time* – The time from sending to the response of a request.
- *Throughput* – The mean number of requests served per time unit.

For example, an SLO could state that the response time should in the mean not be greater than 1 second or that the application should be available 99.9% of the time.

Influencing Performance Metrics

It is important that the SLOs are fulfilled at all times. By measuring the necessary performance metrics and comparing with the corresponding SLOs, it is possible to discover when changes need to be applied to the application deployment. There exist many different types of actions to exert change in a deployment in order to influence performance metrics; here, we will state the most common ones.

Resource allocation. Performance of heavily loaded applications can be improved by assigning more resources. Due to hardware limits, the services that make up the application cannot be deployed with an arbitrary amount.

[7] SLA is a broad concept that can be applied to most provider/user constellations in IT.

For each service deployment, there will be some quantization, but more importantly some upper limit, in the resources allocatable. To circumvent this, it is common to deploy each service as a set of multiple copies, also known as *replicas*. The requests made from one service to another are then *load balanced* between the two sets of replicas, or *replica sets*, according to some strategy in order to effectively utilize the available resources. The replica sets are further beneficial for reliability, as if one replica goes down, the entire application would not cease to function. The resources allocated for a service can then be influenced by scaling its replica set as follows [Qu et al., 2018].

- *Horizontal scaling* – Create a new replica or remove an old replica.
- *Vertical scaling* – Assign or release resources to an existing replica.

Horizontal scaling is arguably the most common, as it is simple to implement in real systems, while vertical scaling serves more a fine tuning purpose.

Where to deploy the replicas is also of importance to, e.g., avoid resource contentions between services requiring the same type of hardware and to minimize communication latency between services. Based on the service model, the application owner may have a more or less fine-grained control over this deployment location, which is otherwise left to the cloud provider. Therefore, a strategy related to horizontal scaling is *service migration*, where a replica is moved by reallocating the resources at a new location. This can, for example, be done to reduce resource contention or to move the service from a suddenly misbehaving machine or to a data center with lower user latency.

Request handling. Instead of changing the resources allocated to an application, it can in some situations be better to change how requests are handled within the application deployment. For example, it is possible to influence performance metrics by the following methods:

- *Load balancer tuning* – The strategy used for request routing between replica sets can be tuned to better utilize the available resources in a changing environment.
- *Admission control* – By prohibiting certain requests from entering the application, the availability can be reduced in order to save some other more important SLO. For example, it can be used as a last resort for applications close to overloading to prevent crashes.
- *Request redundancy* – The response time, especially the tail of its distribution, can be reduced by sending duplicate requests to different replicas and acting on the first response.

The best action to take, be it related to resource allocation or how requests are handled, will depend inherently on a multitude of different factors, such

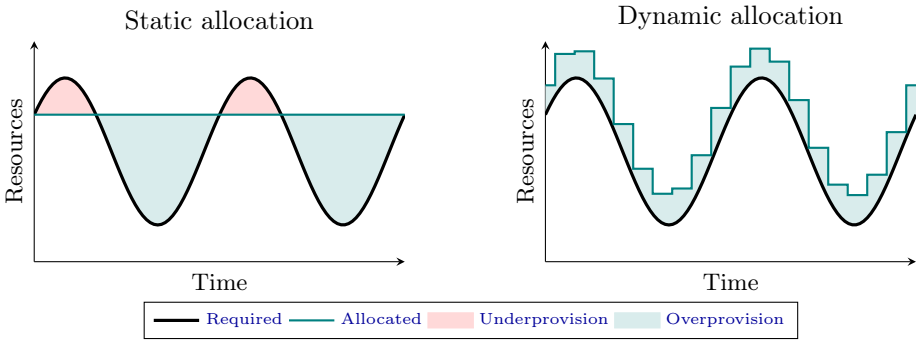


Figure 1.3 A classic illustration comparing static and dynamic resource allocation. Here, the black line shows the required resources to meet the SLOs, which changes in a cyclic pattern to, e.g., accommodate for the changing rates of connecting users during a day. The teal line shows the allocated resources. If this line lies below the required resources, the SLOs will be violated due to resource underprovisioning, indicated by the red areas. If the line instead lies above, we will have overprovisioned the resources, which is is an unneccesary cost. Static allocation potentially results in high overprovisioning, and also SLO violations if we are unable to accurately predict worst-case scenarios. Dynamic allocation on the other hand results in less overprovisioning, ideally it should be zero but it is still needed as there is a limit in resource quantization and how fast we can adapt.

as how "best" is defined, the type and current state of the application, and its SLOs.

Balancing Performance and Cost

The cloud is an inherently dynamic environment. Unpredictable performance can arise from, e.g., varying workloads and resource contentions, both between unrelated processes residing on the same physical hardware and between processes residing in the same service. To cope with this unpredictability in application deployments, it is common to *overprovision* resources to applications in order to avoid accidental violations of SLOs. Thanks to the capability of cloud computing, resources can be dynamically obtained and released when needed, and more optimistic overprovisioning than static provisioning can be performed for the worst-case scenario. A simplistic illustration of these concepts can be seen in Figure 1.3.

Large overprovisioning leads to poor utilization of the hardware, which is costly, both in terms of money and in terms of environmental impact [Buyya et al., 2018]. Keeping this overprovisioning small without violating SLOs is thus of high importance.

Problematic decisions. As discussed, by tracking performance metrics and SLOs, it is possible to discover when some action needs to be performed on the application deployment. But what action should be made in regard to cost efficiency or robustness, for what type of application in what situation, is a difficult and interesting research problem that has historically received a lot of attention. See, e.g., the survey papers [Ardagna et al., 2014] and [Singh and Chana, 2015] for an overview of this field.

Considering the modern trends for cloud applications, this research problem is of high interest due to the increased difficulties they bring. For example, applications designed for fog/edge computing impose nontrivial constraints in the form of latency limits to end users and site dependent costs. Furthermore, applications of many independent services, e.g., using the microservice architecture, create an environment of many changeable parameters and hidden dependencies between services [Gan et al., 2019].

Connection to Automatic Control. The act of balancing performance and costs can be thought of as a control problem and has seen much research considering the broader field of self-adapting software systems [Filieri et al., 2017]. Briefly explained, an application deployed in the cloud can be seen as a dynamical system, with measurements defined via relevant performance metrics, and which can be influenced via control signals by, e.g., scaling resources or tuning load balancers. The balancing act then reduces to designing a controller that via the defined control signals allows the performance metrics to track their SLO defined setpoints while adequately rejecting disturbances, such as workflow fluctuations.

Performance Modeling

How available deployment decisions affect important performance metrics is a nontrivial and application specific question. To make informed decisions, it is important that these connections are known. This is commonly achieved by deriving some sort of *performance model* of the cloud application [Balter, 2013]. The model can then be used to, for example, design strategies for which decision to make in what situation to adequately fulfill the SLOs.

Modeling using queueing theory. Performance models of cloud applications are often based on queueing theory, as the processing of a stream of requests that shares the available computational resources essentially acts as a queueing system. Furthermore, these queues have direct equivalences to important performance metrics.

In queueing theory, a *queue* [Balter, 2013] is defined as some set of servers[8] that serve a stream of arriving requests. The stream is characterized by an

[8] The term "server" is ambiguous, as it is used to describe objects of similar characteristics in both computing and queueing theory. It is important to keep this in mind to avoid confusion.

interarrival time distribution, defining the times between arrivals. In addition, each request is assumed to require some time to be served, defined by a *service time* distribution. Once a request has been served, it departs the queue. The requests present in the application might outnumber the available servers which then need to be shared in some manner. How this is done is defined by the *queueing discipline*, and considering modeling of cloud applications, the most common are

- *First-come first-served* (FCFS) – On arrival, each request is exclusively assigned to a free server. If no server is free, the requests are put in a waiting line, i.e., *queue*, based on the order of arrival. This is the most commonly used discipline in general queueing theory, and what we in everyday language might refer to as a queue.

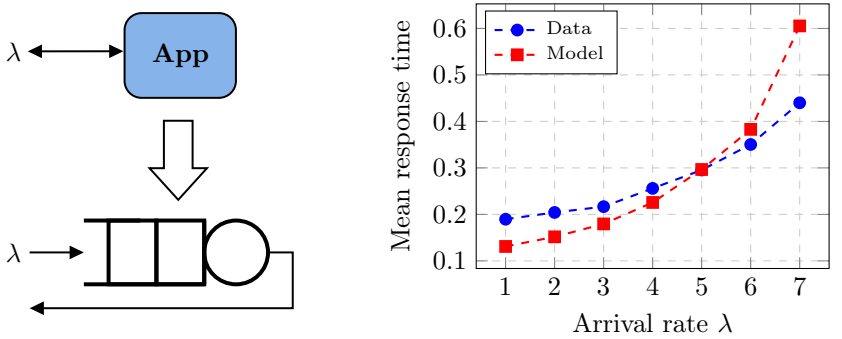
- *Processor sharing* (PS) – The requests share the available servers and are thus served at a speed proportional to the inverse number of requests present in the queue. This discipline is motivated by how CPUs serve multiple processes simultaneously.

- *Delay* (INF) – Each request is assumed to always have a free server to which it can be assigned directly and exclusively. No actual server sharing is performed, and it can be thought of as a queue with infinite servers, hence the acronym.

As a simple example, we constructed a monolithic cloud application performing face detection on submitted images. We then tried to model this application as a simple FCFS queue with a single server and exponentially distributed interarrival and service times, normally denoted an M/M/1 queue. The application was loaded with requests with exponential interarrival times, also known as *Poisson arrivals*, with the rate $\lambda = 5$ and relevant data logged to fit the M/M/1 queue model. The fitted model was then used to predict mean response times over a changing arrival rate. The results seen in Figure 1.4 shows that this simplistic queue model manages to capture the general behavior of the mean response time, but not its values.

Applications do not have to be modeled as a single queue; instead, granularity can be improved by, e.g., using multiple queues in a network, i.e., a *queueing network*, to model dependencies between different resources and/or services in the application. In addition, queues are sometimes allowed to serve multiple classes of requests, where each class has its own service time distribution and on-completion routing destination, but shares the same servers.

Given a queueing model, closed-form expressions of important performance metrics only exist for certain types of queueing disciplines, interarrival and service time distributions. Instead, performance metrics are often obtained via either simulation or approximative methods. Considering



(a) Modeling the application as a single M/M/1 FCFS queue.

(b) The resulting response times from data and model prediction.

Figure 1.4 The simple cloud application, modeled as a M/M/1 queue and loaded with Poisson arrivals with rate λ. The parameters of the queue model are fitted at $\lambda = 5$.

stationary mean values, there exist numerous approximations both for the single queue and queueing network case. One of the most well-known is the *mean value analysis* (MVA) method for obtaining stationary results on queue length, response time, and throughput for certain types of queueing networks, known as *product-form* networks [Bolch et al., 2006]. Another method is the *fluid model*, which models the mean dynamics of the queue lengths as a system of *ordinary differential equations* (ODEs). Obtaining this system of ODEs is in general difficult, but the fluid model requires no product-form assumptions and yields both transient and stationary solutions.

Obtaining a queueing model. Given a cloud application, it is not trivial to derive a queueing model that can adequately capture important performance metrics. Models can be obtained by hand, either directly or via translation from other handcrafted software models, but this takes time and requires expert knowledge of the system. Furthermore, for online purposes, the model might change over time and thus need to be continuously updated. Hence, how to extract queueing models from a running system that can adapt to system changes online is an important research question.

Ideally, the usage of the resulting queueing model should be taken into account when creating it. Certain evaluation methods, such as MVA or the fluid model, require constraints on, e.g., the type of queueing disciplines and distributions to be enforced. This is in general not a problem for evaluation using discrete event simulation, but this can instead be prohibitively expensive for online evaluation, especially considering larger queueing network models.

1.3 Scope and Contributions

The wider scope of this thesis is the study of performance modeling of cloud applications using queueing theory. We adopt the view of a service provider managing a single application. Hence, modeling the underlying infrastructure or multiple applications competing for resources is not explicitly considered. This is a well-studied field, and our niche lies in approximations using the fluid model and how such models can be obtained from modern applications comprised of graphs of smaller services, possibly deployed in multi-cloud environments. The thesis also considers static modeling of request redundancy.

The contributions of the thesis can be split into three main parts.

1. Chapter 3 considers improvements to the fluid model approximation of the mean queue length in networks of PS and INF queues.

2. Chapter 4 and Chapter 5 show how this improved fluid model can be applied to performance modeling of microservice applications and how such a model can be used to balance cost and performance.

3. Chapter 6 considers modeling of request redundancy using the novel concept of *synchronized service*.

The specific technical contributions of each part are summarized below.

1. Improving the mean-field fluid model for PS queueing networks. The fluid model potentially allows for quick analysis of both transient and stationary performance metrics. Unfortunately, deriving an adequate fluid model is generally difficult. Recently, the *mean-field approximation* was used to find a fluid model that is exact in the limit for multi-class queueing networks of PS and INF queues with general *phase-type* service time distributions and no arrivals or departures of requests.

In Chapter 3, this result is extended to allow for arrivals and departures of requests. A compact matrix form for mean-field fluid models of PS networks is introduced and used to prove that the mean queue lengths of these networks converge under normalization to the introduced fluid model when the initial state, arrivals, and servers per queue scale to infinity.

Furthermore, mean-field approximations are known to experience errors when this scaling is small, and there exist refinement methods for general systems. By specifically focusing on networks of PS and INF queues, a simple yet accurate data-driven refinement method based on smoothing of the processor sharing function is introduced. Compared to previous methods, this smoothed fluid model requires no introduction of extra states and is thus computationally cheap even for very large models.

Finally, using this refinement, a closed-form approximation of the entire response time CDF over almost any subset of classes in the network is

introduced, allowing the modeling of response time percentiles. Previously, this was solved by running an extended state simulation of the fluid model. The refinement and response time approximation introduced are found to be accurate across a series of simulation experiments.

2. Using the fluid model to model and manage applications. With the modern trends in cloud computing, modeling and managing cloud applications is an increasingly difficult endeavor. It is desirable to obtain application models via automatic extraction, but current methods are often intricate and yield large and complex models that are unsuitable for online evaluation.

In Chapter 4, a fluid model for microservice applications whose services are potentially distributed over different sites with in-between latencies, such as deployments in fog/edge computing, is introduced. It is based on an intermediate queueing network model using only PS and INF queues, for which the aforementioned smoothed mean-field fluid model can be applied. The model is simple yet general, and is quick to completely extract at application runtime in a distributed fashion given data commonly gathered in distributed tracing setups. This enables quick evaluation of important performance metrics, most notably response time percentiles, and how they are affected by deployment changes.

Further, Chapter 4 also introduces the *federated application* (FedApp) sandbox for experimentation on applications in modern multi-cloud deployments. The sandbox is implemented as clusters of virtual machines on OpenStack[9], a cloud computing platform for IaaS, with the possibility of emulating desirable cluster-to-cluster network characteristics. On top of each cluster, the popular Kubernetes container orchestration tool[10] is deployed along the Istio service mesh[11] for simplified cross-cluster service communication. On top of this sandbox, the introduced model is tested on a small microservice application and is shown to be accurate in most cases.

Later, Chapter 5 demonstrates how automatic differentiation over the smoothed mean-field fluid model can be used to optimize a running cloud application. A controller is designed to perform online tuning of load balancing parameters for minimizing some holistic cost function under constraints to the response time percentiles. This is more general than existing methods for optimizing load balancing parameters but does not give guarantees on neither convergence, speed, nor feasibility. The controller is experimentally evaluated on a simple application distributed across multiple sites with different costs and latencies, and shown to minimize a cost function whilst adhering to the SLO constraints.

[9] https://www.openstack.org/
[10] https://kubernetes.io/
[11] https://istio.io/

3. Analysis of redundant requests using synchronized service. Request redundancy is a commonly used tool for increasing reliability and computation speed in cloud applications with high uncertainty. Copies of requests can either be sent immediately, known as request *cloning*, or they can be sent after some delay, known as *speculative execution*. Modeling of these concepts using queueing theory has traditionally been focused on specific distributions for the interarrival and service time distributions and the FCFS discipline.

In Chapter 6, a new simplified method is introduced to model cloning, based on the concept of synchronized service between request clones. It allows certain cloning systems over n computing servers, each modeled with a single queue, to be equivalently represented as a single queue. The method requires no assumption on interarrival and service time distributions, only that the queueing disciplines are homogeneous and deterministic. For these systems, existing methods for evaluating single queues can then be used to analyze the entire cloning system.

The introduced model is thus more general than existing work, but simultaneously also more narrow in the assumptions needed to obtain synchronized service. Focusing on the PS discipline, we further study how synchronized systems are affected by relaxing these assumptions. Specifically, the effect of delays in the arrival and cancellation of clones is first considered, and approximate upper bounds on the mean response time are introduced. Then, the effect of allowing free allocation of clones over the servers according to some load balancing policy is studied. Finally, this is extended to derive an approximate model for speculative execution. Via simulations, it is shown that the induced errors from arrival/cancellation delays are small and that free cloning and speculative execution behind the popular load-balancing policy JSQ behaves as a synchronized system.

1.4 Publications

This thesis is written as a monography, and in this section the papers that it is based upon will be presented. How the papers connect to the introduced contributions above and the different chapters in the thesis, and the particular contributions made by the authors will be discussed.

Ruuskanen, J., T. Berner, K.-E. Årzén, and A. Cervin (2021a). "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing". *Performance Evaluation* **151**, p. 102231. ISSN: 0166-5316. DOI: https://doi.org/10.1016/j.peva.2021.102231.

This paper constitutes Chapter 3. It presents the compact matrix form for mean-field fluid models of PS networks, along with the convergence proof. The paper further presents the simple and computationally cheap refinement method for the mean-field approximation of the mean queue length dynamics of these queueing networks. Finally, the closed-form approximation of the response time CDF is introduced. The introduced mean-field refinement and the response time approximation are tested on two smaller running examples and one larger experiment on a simulated cloud application.

The ideas, proofs, and simulations were all developed by J. Ruuskanen with discussions and input from A. Cervin. The manuscript was written by J. Ruuskanen with valuable comments from all coauthors.

Ruuskanen, J. and A. Cervin (2022). "Distributed online extraction of a fluid model for microservice applications using local tracing data". In: *IEEE 15th International Conference on Cloud Computing (CLOUD '22)*, pp. 179–190. DOI: 10.1109/CLOUD55607.2022.00037.

This paper forms the basis of Chapter 4. It introduces a simple, yet general, fluid model for microservice applications and shows how it can be completely extracted at runtime from common request tracing data in a distributed fashion. Experiments are run on the FedApp sandbox for a small microservice application of 3 services, deployed over 2 clusters, and it is shown to accurately both model the current operating condition and predict previously unseen conditions.

The ideas, formalization, and experiments were all developed by J. Ruuskanen with discussions and input from A. Cervin. The manuscript was written by J. Ruuskanen with valuable comments from A. Cervin.

Ruuskanen, J., H. Peng, A. Åkesson, L. Larsson, and M. Kihl (2021b). "FedApp: a research sandbox for application orchestration in federated clouds using OpenStack". *arXiv preprint*. DOI: 10.48550/ARXIV.2109.01480.

This paper forms a section in Chapter 4. It is a white paper that introduces the FedApp sandbox, and discusses the different sandbox design choices made to promote easy usage/deployment and to enable cluster-to-cluster network characteristic emulation. The paper further discusses the different sandbox functionalities, how it can be used, and showcases a small application deployed over multiple clusters.

The original tool was developed by J. Ruuskanen, H. Peng, and A. Åkesson with inputs from L. Larsson. All three main developers where initially involved in all parts, but with an extra focus on monitoring by J. Ruuskanen, networking by H. Peng, and the example application by A. Åkesson. After its initial release, the work was continued by J. Ruuskanen. The manuscript was written by J. Ruuskanen with input from the other authors, particularly H. Peng.

Heimerson, A., J. Ruuskanen, and J. Eker (2022). "Automatic differentiation over fluid models for holistic load balancing". In: *IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS '22), to appear.*

This paper forms the basis of Chapter 5. It introduces the cost minimizing controller for load balancing parameters by automatic differentiation of the microservice fluid model introduced in [Ruuskanen and Cervin, 2022]. The controller is experimentally verified using a simple distributed microservice application deployed on the FedApp sandbox, and found to reduce costs under disturbances while adhering to percentile constraints.

A. Heimerson and J. Ruuskanen are both first authors of this work, who together set out to explore model-based optimization of a microservice application. The initial idea of using automatic differentiation was suggested by A. Heimerson, while the microservice fluid model to optimize load balancing weights was suggested by J. Ruuskanen. The procedure was then realized by both first authors with input from J. Eker and formalized by J. Ruuskanen with input from A. Heimerson. Both first authors were involved in the experiments, where A. Heimerson constructed the online controller, and J. Ruuskanen the experimental setup and online model fitting adapted from the setup in [Ruuskanen and Cervin, 2022]. The manuscript was written by A. Heimerson and J. Ruuskanen together with J. Eker.

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020a). "Modeling of request cloning in cloud server systems using processor sharing". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, Edmonton AB, Canada, pp. 24–35. ISBN: 9781450369916. DOI: 10.1145/3358960.3379128.

This paper forms the basis of Chapter 6. It introduces the concept of *synchronized service* and the subsequent equivalent single-queue model for request cloning. The paper further studies two simple systems that fulfill the assumptions for synchronized service. In particular, codesign between two load balancing policies to clusters of servers and the number of clones is considered. In addition, the paper studies the effects when the necessary assumptions are relaxed under the PS discipline. It introduces bounds[12] on the mean response time in the presence of delays in arrivals and cancellation of clones and analyzes the effects of freely choosing servers according to some load balancer. A discrete event simulator is constructed, and it is found that the arrival/cancellation delay bounds are tight for low system loads and that cloning under the JSQ load balancer yields a near-synchronized system.

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 coauthors, 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. The manuscript was written by T. Nylander and J. Ruuskanen, with inputs and comments from the other coauthors.

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020b). "Towards performance modeling of speculative execution for cloud applications". In: *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, Edmonton AB, Canada, pp. 17–19. ISBN: 9781450371094. DOI: 10.1145/3375555.3384379.

This paper forms a section in Chapter 6. It builds on the synchronized service modeling principles from [Nylander et al., 2020a], and extends the model to speculative execution in cloning systems with PS queues behind the JSQ load balancer. In an illustrative example simulation, this model is shown to be accurate.

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 coauthors. 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 [Nylander et al., 2020a]. The manuscript was written by T. Nylander and J. Ruuskanen, with inputs and comments from the other coauthors.

[12] Due to an error in Lemma 4, the bounds have to be viewed as approximate.

Publications Not in Thesis

The author of this thesis has further made contributions to the following papers, mainly in the field of event-based particle filtering, which are not included in this thesis.

Event-based particle filtering. At the start of his PhD studies, the author of this thesis explored particle filtering under event-based sampling with the goal of applying it to state estimation in server systems. As time passed, it was deemed difficult to adequately fulfill this goal, and a decision was made to shift the focus from these algorithms to more general performance modeling in cloud settings. The following contributions made to the field of event-based particle filtering are thus left out in order to make this thesis more streamlined and cloud focused.

Ruuskanen, J. and A. Cervin (2018). "Internal server state estimation using event-based particle filtering". eng. In: *Proceedings of the 4th International Conference on Event-Based Control, Communication, and Signal Processing (EBCCSP '18)*. URL: https://lup.lub.lu.se/search/files/49609380/wip_ebccsp_18.pdf.

Ruuskanen, J. and A. Cervin (2019). "Event-based state estimation using the auxiliary particle filter". In: *18th European Control Conference (ECC '19)*, pp. 1854–1860. DOI: 10.23919/ECC.2019.8796091.

Ruuskanen, J. and A. Cervin (2020). "On innovation-based triggering for event-based nonlinear state estimation using the particle filter". In: *European Control Conference (ECC '20)*, pp. 1401–1408. DOI: 10.23919/ECC51009.2020.9143748.

Ruuskanen, J. and A. Cervin (2022). "Improved event-based particle filtering in resource-constrained remote state estimation". *arXiv preprint*. DOI: 10.48550/ARXIV.2209.14081.

Other publications.

Ruuskanen, J., H. Peng, and A. Martins (2019). "Latency prediction in 5G for control with deadtime compensation". In: *Proceedings of the Workshop on Fog Computing and the IoT (IoT-Fog '19)*. Association for Computing Machinery, Montreal, Quebec, Canada, pp. 51–55. ISBN: 9781450366984. DOI: 10.1145/3313150.3313227.

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

1.5 Thesis Outline

This thesis consists of 7 chapters. Next, in Chapter 2 the necessary background to understand the main parts of this thesis is introduced, along with extensive discussions on the connection between the contributions of this thesis and previous published research.

In Chapter 3, the improvements for the mean-field fluid model of PS queueing networks are presented. This fluid model is later used in Chapter 4 to guide the construction of the intermediate queueing network model for distributed microservice applications. Later in this chapter, the FedApp sandbox is introduced and used to experimentally verify the microservice fluid model. Furthermore, in Chapter 5 this fluid model is used to design the cost minimizing controller for load balancer parameters using automatic differentiation. In Chapter 6, the modeling of request cloning and speculative execution using the concept of synchronized service is presented.

Finally, in Chapter 7 the thesis is summarized, the most important results are discussed, and some directions for future work are provided.

2

Background and Related Work

This chapter gives a more in-depth background and related work connected to the different chapters and contributions of the thesis.

Outline. In Section 2.1, a quick overview of the queueing theoretical concepts necessary to understand the remaining thesis is introduced. Furthermore, Section 2.2 first introduces queueing networks and, subsequently, the mean-field approximation for obtaining fluid models. Section 2.3 then discusses modeling of cloud applications using queueing theory and how these models can be obtained from a running application. Section 2.4 discusses how these models can be used to optimize a running application, in particular considering online tuning of load balancing parameters. Finally, in Section 2.5 the concept of request redundancy is introduced in greater detail.

2.1 Queueing Theory for Performance Modeling

A common methodology for performance modeling of computing systems is to use queueing theory to model the service of multiple requests competing for the same software or hardware resources [Bolch et al., 2006; Balter, 2013].

Here, a finite resource is modeled as a set of k *servers* to which the arriving requests need access for some time to complete its service. At times, the available servers might be fully occupied, and thus different forms of queueing or server sharing need to be enforced to decide when and how the present requests can access the limited resource. For request r, let t_r^a denote its arrival time and t_r^d its departure time from the queue. Furthermore, let t_r^s be its necessary service time. See Figure 2.1 for an illustration.

Requests arrive to the queue according to some *arrival process*, characterized by the interarrival time between subsequent requests. The interarrival and service times are often modeled as stochastic variables, following some

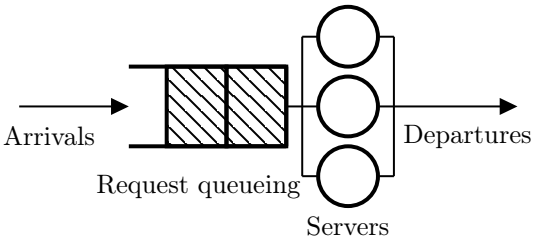


Figure 2.1 Illustration of a single queue with 3 servers. For a request r, the time of its arrival is denoted t_r^a and the time of its departure t_r^d.

distributions

$$\begin{aligned} t_r^a - t_{r'}^a &\sim G^a \qquad \text{(interarrival time distribution)}, \\ t_r^s &\sim G^s \qquad \text{(service time distribution)}, \end{aligned} \tag{2.1}$$

for every request r, where r' is the request that arrives before r.

Stochastic interarrival and service times imply that the total time a request spends in the queue, known as the sojourn time or *response time* T, becomes a random variable, and that the number of requests present in the queue, known as the *queue length* X, becomes a stochastic process. Given knowledge of the arrivals and departure times, the response time of a specific request r can be obtained as $T_r = t_r^d - t_r^a$. Furthermore, the queue length at time t can be obtained as the number of requests that has arrived but not yet departed

$$X(t) = \sum_r \mathbb{1}\left(t_r^a < t\right) - \mathbb{1}\left(t_r^d < t\right), \tag{2.2}$$

where $\mathbb{1}(e)$ is a function returning 1 if the expression e is true, otherwise 0.

Arrivals and service speeds are often characterized by their respective *rate*, which denotes the mean number of requests that arrive or can be served each second, i.e.,

$$\begin{aligned} \lambda &= 1/\mathbb{E}\left[G^a\right] \qquad \text{(arrival rate)}, \\ \mu &= 1/\mathbb{E}\left[G^s\right] \qquad \text{(service rate)}. \end{aligned} \tag{2.3}$$

Common Queueing Disciplines

The way requests present in the queue share the limited servers is determined by the *queueing discipline*. Many different forms of disciplines exist in literature, and the choice depends on the type of system the queueing model is to represent. Considering performance modeling of cloud systems three commonly used disciplines, due to their simplistic yet often adequate description

of scheduling in hardware and software resources, are *first-come first-served* (FCFS), *processor sharing* (PS) and *pure delay* (INF) [Ardagna et al., 2014].

The FCFS discipline is the most common in general queueing theory, where requests are assigned to servers in the order in which they arrive. If no server is available at the time of arrival, the request will wait until its turn. In computing systems, this discipline can be used to model exclusive, nonpreemptable access to a resource, for example requests of equal priority trying to access a disc.

For the PS discipline, it is on the other hand assumed that requests present in the queue share the servers equally at all times. It is furthermore often assumed that the requests are not parallelizable, meaning that each request is served by at most a single server. The speed of which a request is served under this discipline is at any time t scaled with a factor known as the *processor share*, described by the following function of the queue length

$$g\left[X(t)\right] = \frac{\min\left[k, X(t)\right]}{X(t)}, \tag{2.4}$$

where k is the number of servers present in the queue. In computing systems, the PS discipline is commonly used to model time sharing in CPUs, where multiple tasks of the same priority are periodically served in time slices in a round-robin fashion. By assuming that the length of each time slice goes to 0, and that the overhead to switch between tasks is nonexistent, this time-sharing can be idealized as the PS discipline.

Finally, the INF discipline assumes that every request instantly get access to an exclusive server, hence, an INF queue has no resource sharing and can be seen as having an infinite amount of servers. In computing systems, this discipline is used to model behavior and situations where requests are subjected to some delay but where no contention occurs, e.g., a pool of connecting clients or network delays.

Evaluating Performance Metrics From a Queue Model

Given knowledge of the arrival and service rates and the number of servers, two important performance metrics that can be obtained are the *utilization* and the *throughput*. The utilization ρ describes the fraction of time the servers in a queue are occupied, and can be expressed as

$$\rho = \frac{\lambda}{k\mu}. \tag{2.5}$$

As ρ moves closer to 1, both the stationary mean and the variance of the queue length will grow. But as long as $\rho < 1$, these will be finite as the queue will in average process requests faster than they arrive, and the queue is denoted as *stable*.

Furthermore, the throughput τ describes the mean number of requests that complete their service per given time unit and can for stable queues be expressed using the utilization ρ as

$$\tau = k\mu\rho. \tag{2.6}$$

As can be seen, for stable queues the throughput equals the arrival rate λ.

In contrast, queue lengths and response times are more complex to obtain, as they depend on the interarrival and service time distributions, as well as the queueing discipline. Before delving deeper, we first introduce the following common way of classifying a queue based on its properties.

Definition 2.1—Kendall's Notation
A queue can be classified with the notation X/Y/k–Z where (i) X, Y are the types of the interarrival and service time distribution, respectively, (ii) k the number of servers, and (iii) Z the queueing discipline.

A common distribution type is M (Markovian), which implies that the corresponding distribution is exponential, while G (General) implies that no assumption, apart from positive support, on the distribution is made. The Kendall notation can also include additional queueing properties not considered in this thesis.

It is always possible to obtain the queue lengths and response times of a given queueing model using discrete event simulation, but in some cases closed-form solutions or approximations exist. A common example is the M/M/1–FCFS or M/G/1–PS queue, for which the stationary mean queue length becomes [Balter, 2013]

$$\mathbb{E}[X] = \frac{\lambda}{\mu - \lambda}. \tag{2.7}$$

For these, the entire stationary queue length distribution is in fact known, where $\mathbb{P}(X = n) = \rho^n(1 - \rho)$. However, this is not commonplace; in most cases, one has to settle for some approximation of the stationary mean.

Queue lengths and response times are closely related, as captured by *Little's law*, which states that the stationary mean queue length is equivalent to the product between the arrival rate and the mean response time, i.e.,

$$\mathbb{E}[X] = \lambda\mathbb{E}[T]. \tag{2.8}$$

Most notably this relationship holds independently of the distributions, number of servers and queueing disciplines [Little, 1961].

For other response time metrics, such as different α-percentiles φ_α, such tidy formulas rarely exist for the arbitrary queue. In a few cases, the response time distribution has a known closed-form solution from where metrics can be

derived. For example, revisiting the M/M/1-FCFS queue, its response time CDF is given by $\Phi(t) = 1 - e^{-(\mu-\lambda)t}$ [Bolch et al., 2006] and thus the φ_α can be obtained as the time t such that $\Phi(t = \varphi_\alpha) = \alpha$. More generally, given the mean response time, φ_α can be bounded from above using the Markov inequality to derive $\varphi_\alpha \leq \mathbb{E}[T]/(1-\alpha)$. This bound can however be quite loose as shown in the initial motivating example in [Pérez and Casale, 2017].

The fluid model. An alternative to closed-form solutions of stationary queue lengths is instead to try to approximate the time evolution of the entire queue length distribution using a *diffusion approximation*, or the time evolution of the queue length mean with a *fluid model*. A clear benefit of such a model is that it can be used to obtain *transient* metrics as well as stationary ones. But, as with closed-form expressions, it is generally hard to find a good diffusion approximation or fluid model given a general queue, as made evident by the attention the topic has historically received [Schwarz et al., 2016, Section 3.3].

In particular, the fluid model expresses the time evolution of the mean queue length as the flow in and out of some state x, which can be formulated as the following ODE:

$$\frac{dx}{dt} = -q^{out}(x) + q^{in}. \tag{2.9}$$

In the single queue case, $q^{in} = \lambda$ and $q^{out}(x)$ is a function relating the current mean queue length to the rate of which requests depart the queue.

One general method of finding a suitable $q^{out}(x)$ is the pointwise stationary fluid flow approximation (PSFFA) [Wei-Ping Wang et al., 1996]. Here $q^{out}(x) = \mu G^{-1}(x)$, where $G(\rho) = \mathbb{E}[X]$ is some invertible function relating the utilization to the stationary mean queue length. This ensures a stationary solution of (2.9) at $x = \mathbb{E}[X]$, as the flow in this case becomes $\dot{x} = -\mu\rho + \lambda = 0$. In particular, for M/M/1 queues, the PSFFA model is obtained by inverting (2.7), which yields the well-known Tipper model [Tipper and Sundareshan, 1990]

$$\frac{dx}{dt} = -\mu\frac{x}{x+1} + \lambda. \tag{2.10}$$

2.2 Queueing Networks

Considering a computing system consisting of multiple interacting parts, such as systems often observed in cloud computing, model granularity can be improved by representing the different resource components and their interactions as a *network of queues* [Bolch et al., 2006]. In the network, when a request is fully served in a queue it is either routed to a new queue, or it

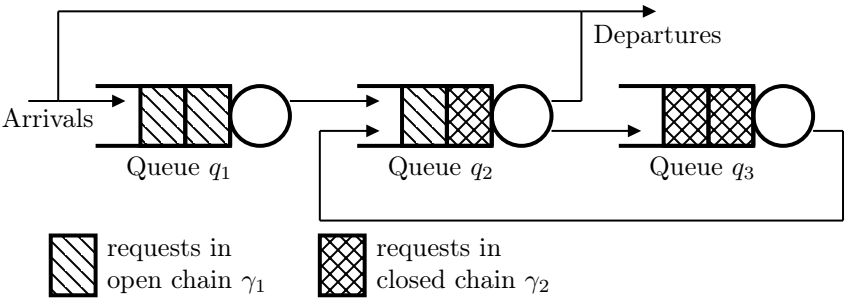


Figure 2.2 An example of a mixed queueing network with three queues and two chains. q_1 and q_3 each holds a single class, c_1 and c_4, while q_2 holds the two classes, c_2 and c_3. Completed requests in (q_1, c_1) are routed to (q_2, c_2), which in turn routes requests either back to (q_1, c_1) or out from the network. New external requests can further arrive to (q_1, c_1). For (q_2, c_3), completed requests are instead routed to (q_3, c_4), and from there back to (q_2, c_3). This forms two disjoint chains, one open, $\gamma_1 = \{(q_1, c_1), (q_2, c_2)\}$, and one closed, $\gamma_2 = \{(q_2, c_3), (q_3, c_4)\}$.

departs the network entirely. Requests can also arrive to queues from outside the network. Each queue is allowed to have its own queueing discipline, service time distribution and number of servers. Arrivals are however both dependent on the existence of external arrivals to this queue, and the routing from other queues. Further, the queues in a network are often allowed to have multiple *classes* of requests that it can serve, each with its own service requirements and on-completion routing destinations.

The paths that requests can travel over the network via the class-to-class routing between queues form disjoint subnetworks over the request classes, known as *chains*. As the chains are disjoint, there exists no path in the network from any class in a chain to any class in any other chain. Further, every class in every queue will belong to exactly one chain. A chain that contains classes to which requests can arrive to the network from external sources, and from where requests can depart the network is known as an *open chain*. Similarly, a chain that contains no such classes is known as a *closed chain*. The absence of arrival and departures implies that the population of request will be constant over time in closed chains, while the population in open chains can vary.

A queueing network consisting only of open or closed chains is termed an *open* or *closed network* respectively, while a network allowed to contain both open and closed chains is known as a *mixed network* [Baskett et al., 1975]. In performance modeling for cloud systems, these mixed networks are of interest to study as it allows for modeling of advanced behavior such as component

interactions in software layers subjected to both synchronous (blocking) and asynchronous (nonblocking) calls [Franks et al., 2009]. An illustration of a mixed network is presented in Figure 2.2.

Evaluating a Queueing Network

Evaluating different performance metrics of a queueing network is in general a more difficult procedure than for the single queue case, as arrivals to every queue can be dependent on both external arrivals and departures from the other queues. As for single queues, it is always possible to directly estimate the metrics given a queueing network via discrete event simulation. However, this can be prohibitively computationally expensive for many use cases, such as real-time evaluation for online decision making, as the number of states can be very large. A more tractable approach is to instead rely on approximation methods of important performance metrics, and in certain cases results from the single queues can be utilized.

Considering an open network of m M/M/1 queues, an example of a so-called *Jackson network* [Jackson, 1957], the total probability of finding all n_i requests in every queue i is given by

$$\mathbb{P}\left(X_1 = n_1, X_2 = n_2, \dots\right) = \prod_{i=1}^{m} \mathbb{P}\left(X_i = n_i\right) = \prod_{i=1}^{m} \rho_i^{n_i} \left(1 - \rho_i\right). \tag{2.11}$$

Interestingly, the probability of finding the entire network in a certain state is given as the product of the probabilities for the single queues. Queueing networks with this property are commonly known as *product-form* networks.

Extensive research has been conducted on which types of queueing network are classified as product-form networks. A most notable result is the so-called *BCMP theorem* [Baskett et al., 1975], which states that multi-class, mixed networks where each queue is allowed to be any of -/M/k-FCFS, -/G/1-PS, -/G/∞-INF among others, have this product-form property. There exist a number of methods to evaluate the steady state performance metrics for product-form networks, arguably the most well-known is the mean value analysis (MVA) [Reiser and Lavenberg, 1980], which simultaneously obtains the throughput, mean response time, and mean queue length for every queue. We refer to [Bolch et al., 2006, Section 7.3] for a more detailed description of both the types of networks that classify as product-form and the methods available to solve them.

For queueing networks not in product-form, or if transient performance metrics are wanted, one could instead try to find either a diffusion approximation or a fluid model for the entire network, in order to approximate the intractable dynamics of the queue length distributions or expected values. In the case of performance modeling for cloud systems, fluid models in par-

ticular are of high interest to study, as the time evolution of mean request population in every queue reduces to solving a set of ODEs.

These sets of ODEs are quick to solve, even for large models given modern methods, and are of importance when devising control approaches [Filieri et al., 2017]. Furthermore, from the resulting vector of modeled queue lengths $\boldsymbol{x}$, both transient and stationary values can be obtained for important performance metrics [Pérez and Casale, 2013]. Assuming stability and a probabilistic routing policy where $P_{j,i}$ gives the probability of on completion routing from queue j to i, the throughput at queue i is obtained as the sum of external arrival rates and inflows from other queues

$$\tau_i(t) = \sum_j P_{j,i} q_j^{out}\left[x_j(t)\right] + \lambda_i. \tag{2.12}$$

Using the throughput, the utilization can be obtained from (2.6) as

$$\rho_i(t) = \frac{\tau_i(t)}{k_i \mu_i}. \tag{2.13}$$

At stationarity, the mean response time can be obtained from Little's law (2.8) together with the stationary mean queue length and throughput

$$\mathbb{E}\left[T_i\right] = \frac{x_i}{\tau_i}. \tag{2.14}$$

The same formula can also be used to approximate the mean response time at transient times.

However, just as for the single queue case, it is in general hard to derive a suitable fluid model, or diffusion approximation for a given system. Some notable works include; Fluid model and diffusion approximation for state dependent open networks of Markovian FCFS queues [Mandelbaum and Pats, 1998], convergence studies on fluid models for queueing networks with head-of-the-line proportional PS queues [Bramson, 1996], fluid model for layered queueing networks with exponential service times via PEPA algebra [Tribastone, 2013], and diffusion approximation for queues with phase-type service time distribution using Stein's method [Braverman and Dai, 2017].

Response time percentiles. The fluid model allows for a computationally cheap approximation of the response time distribution at stationarity. It builds on fixing the modeled queue lengths at their stationary values and numerically computing how much flow mass of a single newly arrived request remains in the network after some time [Pérez and Casale, 2013].

This is an important property of fluid models, as response time percentiles are common in SLAs but difficult to obtain for queueing networks apart from expensive simulations or inexact Markov inequalities. In a few special

cases, the response time distribution obtains a closed form solution [Harrison and Knottenbelt, 2006]. Furthermore, for certain network types, e.g., product-form networks of PS and INF queues, computationally efficient approximations can be obtained by decomposing the queueing network into a set of Markov processes [Casale, 2010]. Finally, in the more general case that the queueing network can be translated into a so-called quasi-birth-death process, *matrix-analytical methods* [Latouche and Ramaswami, 1999] can be employed to obtain different performance metrics, including response time distributions. However, this is more suitable for considering smaller or single queue models, as the method can be expensive for larger networks.

The phase-type distribution. Many evaluation methods for queueing networks build on Markovian assumptions on the interarrival and service time distributions. To circumvent this and allow for more general distributions, it is common to consider interarrival and/or service times that follow a *phase-type* (PH) distribution. The PH distribution represents its random variable as the time-to-absorption in a single-sink internal Markov chain with S transient states and one absorbing state, and can approximate any other positive support distribution [Balter, 2013]. The transition rates between the transient states are given by the matrix Ψ, while the rates from the transient states to the absorbing state are given by the vector ψ. Arriving requests are assigned to one of the transient states according to the probability vector ζ.

For example, a request that enters a queue with a PH distributed service time will be assigned immediately to one of the internal Markov chain states according to ζ. The request will then move around in this internal chain according to the rates in Ψ and ψ, until it reaches the single absorbing state, where it immediately departs for the next queue or from the network entirely.

The Mean-Field Fluid Model

The *mean-field approximation* [Bortolussi et al., 2013] gives a way to approximate the mean dynamics of a stochastic process as a fluid model. It states that given a special type of continuous-time Markov chain (CTMC), known as a *density-dependent population process*, the time evolution of the mean population in each state can be approximated as the solution of the following system of ODEs:

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= F(\boldsymbol{x}), \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0). \end{aligned} \tag{2.15}$$

Here $F(\boldsymbol{X})$ is the drift function of the CTMC, dependent on the current state population vector $\boldsymbol{X}$. Under certain conditions, it can be shown that the population dynamics of the CTMC converges to its mean-field approximation as a property commonly referred to as *system size* goes to infinity—a

result commonly known as Kurtz's theorem [Kurtz, 1970]. However, it is well-known that mean-field approximations can generate inaccurate results when the system size is small [Gast, 2020], and bounds for these errors have been studied using, e.g., Stein's method in [Ying, 2017]. Based on such results, refinements to the mean-field approximation to improve accuracy have recently been presented in [Gast and Van Houdt, 2017; Randone et al., 2021].

We introduce the following definition to differentiate between (nominal) fluid models that can be obtained by deciding input/output relationships in (2.9), and fluid models that can be derived as a mean-field approximation.

Definition 2.2—the mean-field fluid model
For some types of queueing networks, it is possible to translate the stochastic process driving the queue lengths in the entire network to a density-dependent population process that fulfills Kurtz's theorem. The resulting fluid model, given by (2.15), is called the *mean-field fluid model*.

The mean-field fluid model has been used to examine software systems modeled as queueing networks. For example, [Incerto et al., 2017; Incerto et al., 2018b; Incerto and Tribastone, 2019] used this fluid model to derive a scheme for model-predictive control of server systems. It was further used to study the estimation of service times and transition probabilities using a moving horizon estimator in [Incerto et al., 2018a; Incerto et al., 2021], and using reinforcement learning in [Garbi et al., 2020]. Furthermore, [Vasantam et al., 2018; Kielanski and Van Houdt, 2021] used this fluid model to study the *power-of-d* load balancing policy for systems of PS queues.

Recently, in [Pérez and Casale, 2013; Pérez and Casale, 2017] it was shown that for closed networks, the mean-field fluid model can be obtained for any combination of multi-class PS or INF queues with general service times in the form of PH distributions. Later, in [Zhu et al., 2020], this result was extended to cover discriminatory processor sharing. These results are of great interest for performance modeling of cloud applications, as both the PS and INF queues are common when modeling cloud systems. In Chapter 3, we extend the results from [Pérez and Casale, 2013; Pérez and Casale, 2017] to also cover *mixed* networks. Although open chain behavior can be approximated by closing the chain via the introduction of bottleneck queues [Bolch et al., 2006, p. 507], which for BCMP networks can be shown to converge in the limit [Anselmi et al., 2013], we show that the mean-field fluid model can be obtained directly from a mixed PS network. We also introduce a simple technique based on data-driven smoothing of the drift function, which, in contrast to the existing refined mean-field models, is computationally cheap even at very large scales. Finally, a closed-form approximation for the response time distribution over almost any subset of classes in the network is introduced.

2.3 Cloud Application Modeling

Modern trends such as the microservice architecture [Gan et al., 2019] and fog/edge computing [Yousefpour et al., 2019] make the orchestration of cloud applications more complex. Each service can be scaled individually, the interactions between services can be intricate, and the service graph might span different machines and even different clouds. Thus, new challenges arise for old problems in automatic runtime management, such as balancing between quality of service and cost minimization in these new more complex environments. In the literature, it is common to employ analytical performance models when designing such management methods [Ardagna et al., 2014; Amiri and Mohammad-Khanli, 2017; Moghaddam et al., 2019].

Performance modeling of cloud applications is a vast research area with many different modeling methodologies, as shown in the preceding survey papers. In this thesis, we focus on modeling using queueing theory, especially on queueing networks for which the mean-field fluid model can be obtained. A closely related but more advanced modeling method is the *layered queueing network* (LQN), where service times for queues in a network are given as the response time from another "deeper" queueing network. The queues of this deeper network can in turn have service times given as the response time of even deeper networks, etc. This layering enables the model to capture detailed interactions between software and hardware resources [Franks et al., 2009]. Other methods for performance modeling include, e.g., Petri nets [Brogi et al., 2007], stochastic process algebra [Hermanns et al., 2002], machine learning models [Yanggratoke et al., 2015a] and network calculus [Duan, 2011].

There exist many examples in the literature where queueing models of cloud applications have been used. The choice of queueing model depends both on the type of application and on the context. For example, simpler applications can sometimes be adequately modeled as single queues, as can be seen considering, e.g., web server modeling in [Cao et al., 2003] using a single M/G/1-PS queue with limited queue size, and in [Pacifici et al., 2005] using a single M/M/1 queue. Furthermore, in [Yanggratoke et al., 2015b] a part of the storage backend in Spotify was modeled with parallel M/M/1-FCFS queues, representing access to memory and discs.

Larger applications of multiple tiers are instead commonly modeled as queueing networks, where each tier is represented by a small number of queues. In an early work, [Urgaonkar et al., 2005] modeled a multi-tier application with a closed queueing network, where each tier replica was modeled using a single M/G/1-PS queue. A similar model was later adopted in, e.g., [Xiong et al., 2011] where an open tandem queueing network with each tier modeled as a single M/G/1-PS queue was used to optimize both the mean response time and resource usage. Also, in [Goudarzi and Pedram, 2011] each tier replica was instead modeled by two queues, one for serving requests going

forward and one for returning responses, and used in an optimization scheme to maximize profit while adhering to the SLAs. In [Addis et al., 2013], each tier was instead modeled as a collection of parallel multi-class M/G/1-PS queues, in order to model and optimize over multiple applications that inhabit a cloud service center. Also, in [Kattepur and Nambiar, 2015] each tier was modeled using a tandem queueing network, with queues adhering to a product-form constraint but also allowing for multiple servers. This was done to model, e.g., multi-core CPUs as M/G/k-PS queues, for the purpose of developing an MVA extension to handle such networks. Furthermore, in [Wang et al., 2019] a tandem queueing network was used to model and optimize the size of thread and connection pools between tiers.

Considering the microservice architecture in particular, [Niu et al., 2018] modeled each service as a single M/G/1-PS queue to optimize load balancing between replica sets in the presence of multiple chains using the same services. Furthermore, in [Yu et al., 2019], each service was instead modeled as a M/M/k-FCFS queue and used in an optimization scheme to minimize power consumption and response time by request routing and service placement. Applications deployed in a fog/edge computing environment have also been modeled using queueing networks, e.g., in [Xiao and Krunz, 2017] where each fog node was modeled as a single M/M/1 queue and the resulting network was used to study optimal workload offloading between multiple fog nodes and a cloud data center.

More advanced application models often utilize layered queueing networks for increased accuracy, albeit at the cost of being harder to construct and more expensive to evaluate. Some examples include [Faisal et al., 2013], where an LQN model was used to study the impact of network delays between different parts of an application deployed in a fog/edge computing environment. In [Barna et al., 2017], a model for multi-tier applications was developed using a simple three layer queueing network, where each layer in each tier has a single multi-server queue representing either the corresponding software, container, or hardware. In addition, in [Gias et al., 2019] a LQN model of a microservice application was used to optimize combined vertical/horizontal autoscaling under SLA constraints.

Runtime Model Extraction

Deriving an adequate queueing model for a cloud application is not a trivial endeavor. One straightforward option to try is to craft such a model by hand. Furthermore, queueing models can be obtained from certain architectural models of an application, which are sometimes used during, e.g., application development. For example, considering the Palladio component model [Becker et al., 2009], a layered queueing network model [Koziolek and Reussner, 2008], or a queueing network model [Pérez and Casale, 2013] can

be readily acquired.

However, the handcrafting of either queueing models directly or the transitional architectural models takes time and requires expert knowledge, and it has to be redone each time the application is updated. An alternative is instead to try to automatically extract a model from system logs and tracing data. For example, considering the automatic extraction of architectural models, [Brosig et al., 2011] studied the extraction of application models based on the Palladio component model from system monitoring data collected at runtime. Furthermore, in [Spinner et al., 2019] runtime extraction of combined models of both the application and the underlying infrastructure was studied using an agent-based approach and the Descartes modeling language.

Considering the extraction of a queueing model directly from data, [Harbaoui et al., 2010] proposed a method based on decomposing the application into black box components. Each black box was to be load tested and used to form a repository of M/G/k-FCFS models, which then the complete application model could be derived from. A similar approach was adopted in [Begin and Brandwajn, 2016], but here the queueing models of the components were assumed to be precalibrated. The paper instead focused on the combination of these into a complete application model. In [Desnoyers et al., 2012], a system was proposed to classify requests visiting each software component and the component interactions using data mining techniques on system log data. Each component is then modeled as a multi-class M/G/1-PS queues, and via the inferred interactions stitched together to form a queueing network model. Furthermore, in [Awad and Menascé, 2020] a large framework for automatic model extraction was presented. The framework provides a step-by-step workflow to derive both workload and system models by analyzing system logs and configuration files. A queueing model is then created by load testing and matching the extracted models to a repository.

Automatic construction of LQN models has also been studied. For example, [Hrischuk et al., 1999] developed a method for extracting LQN models of distributed software systems that contain both synchronous and asynchronous calls. The method builds on sampling the system with special annotated traces to create LQN submodels, which are then merged to create the application model. In [Israr et al., 2005], this method was extended to give up the need to annotate traces, but at the expense of capturing the joining of previously forked request paths. Furthermore, [Mizan and Franks, 2011] proposed a framework for efficiently capturing LQN models of distributed systems without the notion of global time. The model is created by examining traces at each node, which in turn are generated by instrumenting the software program.

Existing automatic extraction methods such as these are often complex, rely on offline stages, and produce large scale models. Furthermore, once a

model structure has been established, service time distributions and routing probabilities must be estimated, which is not trivial [Spinner et al., 2015]. Although such methods have the potential to produce accurate models, large and complicated models are detrimental to runtime application management, as the performance model must be repeatedly updated and evaluated. Hence, when designing an automatic extraction method to be used in balancing performance and cost, it is important to take into account the complexity of both the extraction and model evaluation.

In Chapter 4, we introduce a simple yet general queueing network model for microservice applications. The model can be completely extracted at runtime from request tracing data commonly available in software for distributed tracing. It models each service as a multi-class PS queue and each service-to-service latency as an INF queue. Each class is further assumed to have PH distributed service times. This allows the mean queue length dynamics to be quickly evaluated using the smoothed mean-field fluid model introduced in Chapter 3.

2.4 Model-Based Optimization at Runtime

As the cloud is an inherently dynamic environment, balancing performance and cost has to be performed at runtime to ensure that the SLOs are fulfilled at all times. Designing such balancing methods is, however, a challenging research problem due to, e.g., the intricate relationship between workload, actions, and performance metrics. This issue has historically received much attention, and a typical way to tackle it is by posing the cost and performance trade-off as an optimization problem. As can be seen in Section 2.3, this is a common use case for performance modeling of cloud applications. However, the resulting optimization problem often becomes difficult to solve at runtime. Especially considering actions such as service scaling or scheduling replicas across compute instances, which reduces to mixed integer programming or bin packing problems, respectively.

Considering the aforementioned papers in Section 2.3, an example of model-based optimization is [Xiong et al., 2011], where the simple model for multi-tier applications yields a closed-form solution to optimally decide the CPU shares of each tier, in order to minimize the mean response time. A PI controller was then used to decide the total allocated CPU capacity in order to keep the response time at some reference. On the contrary, in [Goudarzi and Pedram, 2011], the slightly more advanced model and problem statement of maximizing revenue over multiple resources reduce to a nonlinear mixed integer problem. An upper bound on profit was derived and a method, similar to gradient descent, was developed to heuristically solve the problem. Furthermore, [Addis et al., 2013] aims to optimize over VM

placement, resource allocation, and load balancing simultaneously. The optimization problem is split over different time scales and solved using heuristic methods such as fixed-point iterations with initial conditions selected using a greedy approach. Regarding microservice deployments, [Yu et al., 2019] considered joint power consumption and mean response time minimization by tuning scaling, request routing, and replica placement. The resulting optimization problem was split into three stages considering each type of action, where each part was solved using heuristics. Similarly, [Gias et al., 2019] considered combined vertical/horizontal scaling to maximize revenue while minimizing allocated CPU capacity under constraints on utilization and response time. The resulting problem was solved heuristically using a genetic algorithm.

Control theoretical methods. Control theoretical methods have also been proposed to handle the runtime optimization of application deployments. In [Incerto et al., 2017], *model-predictive control* (MPC) of a simple queueing network evaluated using the mean-field fluid model was considered. The goal was to optimally track reference values for throughput and queue lengths, by tuning routing and horizontal scaling, and it was shown that the problem of deciding the optimal control signal reduced to a linear mixed integer program that can be quickly solved for the considered small problem. Later, these results were extended in [Incerto et al., 2018b] to also include vertical scaling, different classes of requests, and response time constraints. Furthermore, [Barna et al., 2018] used a *linear-quadratic regulator* (LQR) to perform horizontal scaling of replicas and threads per replica to keep response time at some reference level. The linear model was created by numerical linearization of an extended LQN model of the target application.

Both MPC and LQR are advanced concepts in control, simpler approaches more suitable for theoretical analysis has also been considered. For example, [Robertsson et al., 2004] used PI control and a saturated linear model to devise, and analyse the stability of, admission control for web servers modeled with a single G/G/1 queue. In [Nylander et al., 2018], it was found that naively combining load balancing with graceful degradation using the brownout concept at each replica could produce response times of high variance. To remedy this, a simple control architecture based on multiple pure integral controllers was developed. Furthermore, [Millnert and Eker, 2020] studied combined vertical and horizontal autoscaling by utilizing the different time scales of the two actions to explore a *mid-ranging* control strategy. A feed-forward approach was used for the horizontal scaling, while a PI controller was used for the vertical scaling.

Reinforcement learning. The infeasible nature of the general optimization problem has spawned a recent interest in applying machine learning methods, especially reinforcement learning, to optimally balance performance

and costs. For example, [Wang et al., 2021] studied dynamic coordination of microservice placement in a fog/edge computing setting using reinforcement learning. The goal was to minimize the total migration cost and long term delay, while keeping short term delays constrained, for users moving between edge computing sites. In [Heimerson et al., 2021], the combined problem of workload load balancing and cooling control within a datacenter was studied using reinforcement learning, in order to reduce energy consumption. Furthermore, [Rossi et al., 2019] studied combined vertical/horizontal scaling of microservices using reinforcement learning, where the cost to minimize was a combination of the cost of performing an action, allocation of resources, and violation of the mean response time constraints.

Load Balancing

As optimizing over all possible actions reduces to a very difficult problem, we will in this thesis focus on the subproblem of tuning load balancing parameters between replicas to minimize the running cost of a microservice application while adhering to its performance constraints.

There exist many load balancing strategies to determine where to route new requests. Most strategies are model free approaches, and the "best" choice depends on the characteristics of the workload and the system, as well as the desired outcome [Wang and Morris, 1985]. Arguably, the most commonly used strategies are *round robin* (RR), *random* and *join-shortest-queue* (JSQ) (also known as shortest-queue-first/least-connected/least-requests) and their weighted counterparts [Sharma et al., 2008; Lee and Jeng, 2011]. These strategies are available in many modern software tools, such as the proxies Envoy[1] and Nginx[2]. JSQ has in general better performance than both random and RR, and is considering balancing over PS queues near optimal in minimizing the total mean response time [Gupta et al., 2007b]. However, it requires knowledge about the current state in each replica, and for each decision the replica with the shortest queue length has to be found, which makes JSQ less ideal to implement and run at scale. To handle these drawbacks, many improvements have been suggested, such as *power-of-d* (SQ(d)) [Azar et al., 1994; Bramson et al., 2010], *join-idle-queue* (JIQ) [Lu et al., 2011] and *join-the-best-queue* (JBQ) [Spicuglia et al., 2013].

Optimal probabilistic routing. Although JSQ and its extensions promise a good mean response time in a load balancing scenario, they are difficult to analyze under more general settings and costs. We thus restrict ourselves to study the *weighted random* policy, which allows us to adequately model the microservice application as a queueing network with probabilistic routing.

[1] https://www.envoyproxy.io/docs/envoy/v1.5.0/intro/arch_overview/load_balancing

[2] https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

The problem thus reduces to finding the routing probabilities that minimize cost while adhering to the SLOs.

This problem is well-investigated in the queueing theory community. Regarding open networks, response time minimization has been studied in, e.g., [Fratta et al., 1973] considering M/M/1 FCFS queues with link constraints using flow deviation, [Borst, 1995] considering M/G/1 FCFS queues over a weighted sum of response times, and [Guo et al., 2004] considering G/G/1 queues of either FCFS and PS discipline with constraints on response time variance. For closed networks, throughput maximization has been studied for product-form networks in, e.g., [Kobayashi and Gerla, 1983] using gradient stepping as the gradient is readily obtained from the MVA algorithm, and [Anselmi and Casale, 2013] using closed-form heuristics based on heavy traffic limits. In [Hordijk and Loeve, 2000] it is shown that for product-form networks with a general cost function and constraints on queue states, a Nash equilibrium can be obtained via deterministic routing. To make the resulting optimization problem feasible, these methods typically only consider simple cost functions and performance metric constraints (if any) over specific types of queues and networks.

Similar conclusions can be drawn in the vast body of literature dealing with the tuning of load balancing parameters. For example, in the two aforementioned papers [Addis et al., 2013; Yu et al., 2019], the joint optimization problems over both request routing and resource allocations must be solved heuristically, and only considering average response times as part of the cost function. Further, in [Wang and Casale, 2014] the results from [Anselmi and Casale, 2013] are evaluated for heuristic weighting of the RR policy in order to maximize a linear revenue function of the throughput. In [Javadi and Gandhi, 2017] a load balancer accounting for hidden interference was developed to minimize response time percentiles. The method builds on using an approximate formula for the percentiles and assuming M/M/1 queues, which yields a closed-form solution for weighting the RR policy. Further, [Yao et al., 2022] uses a Kalman filter to track processing speed at servers from measured average response times. This information is then used to heuristically route requests to servers with the shortest expected delay.

In Chapter 5, we take an alternative approach that enables us to tune the weights of multiple load balancers using the random policy under general costs and constraints. The method is based on online gradient stepping, where the gradient is obtained via automatic differentiation of the microservice fluid model from Chapter 4. This allows us at runtime to optimize a microservice application distributed over multiple sites, with arbitrarily defined costs and constraints from any performance metric obtainable from the fluid model, most notably response time percentiles. However, by adopting such a general approach, we forgo any theoretical results on global optimality, feasibility, and convergence speed.

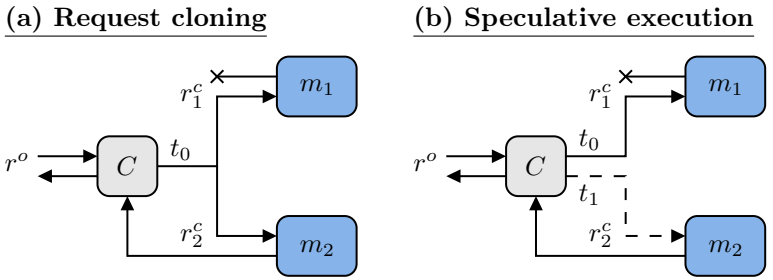


Figure 2.3 Comparison of (a) request cloning and (b) speculative execution over two replicas m_1, m_2 of the same server. At time t_0, a request r arrives at some cloning dispatcher C. In (a), the request gets immediately copied into request clones r_1^c, r_2^c and dispatched to m_1, m_2. The request clone that first finishes being served, r_2^c in this case, gets returned while r_1^c is removed. In (b), r_2^c instead gets dispatched first at t_1 after some speculation time has passed and r_1^c has not yet completed. In this illustration, r_2^c is dispatched and completes before r_1^c.

2.5 Request Redundancy

In cloud computing, *request cloning* refers to the generation of redundant requests. Instead of sending a user request to only one computing server, the request is copied into *clones* and sent to multiple servers. The response to the original request is then set to the result of the server that first completes the processing of its cloned request. When this happens, the pending requests (i.e., the other clones that are queued or still being processed in the other servers) are typically canceled, a policy known as *Cancel-on-Complete (CoC)* cloning. Another less common policy is to cancel the remaining clones once the first clone *starts* its service, known as *Cancel-on-Start (CoS)* cloning [Gardner, 2017].

The motivation for request cloning comes from the desire to reduce the mean and tail response times of applications running in uncertain environments. For cloud applications, the method can be seen as an intuitive way to increase the predictability of responses, and can in certain cases yield a significant improvement in performance [Ananthanarayanan et al., 2013]. However, launching more copies of the same request increases the overall utilization, which leads to a trade-off between reducing response times by multiple requests and increasing response times by increasing the load.

Speculative execution. Cloning can be seen as a special case of *speculative execution*. Here, instead of immediately replicating requests, copies of the original are sent out only after some set waiting time has passed. The

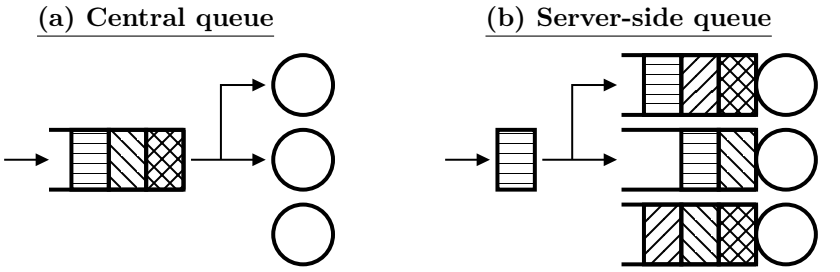


Figure 2.4 Illustration of request cloning using (a) a central queue to dispatch request clones to free servers, and (b) server-side queueing for immediately dispatching clones to servers.

difference between the two concepts is illustrated in Figure 2.3. The general aim of speculative execution is, as for cloning, to increase the response time predictability of requests in uncertain environments. However, by only acting on the slow running requests, the effect of these can hopefully be mitigated without incurring the cost and load increase of cloning every request. Speculative execution is particularly popular in big data frameworks such as MapReduce [Dean and Ghemawat, 2008] or Spark [Zaharia et al., 2012].

Modeling Redundancy Using Queueing Theory

Recently, there has been an upsurge in interest in modeling applications subjected to cloning and speculative execution. Most approaches adopt queueing theory in some form, often considering either central queueing or server-side queueing, see Figure 2.4 for an illustration.

Cloning can be seen as a particular case of the (n, k) fork-join model, where a request is split into n subtasks that are distributed to servers. The request completes when at least $k \leq n$ of those tasks are completed. Cloning implies that the n subtasks are identical and $k = 1$. Approximate analysis and latency bounds have been extensively studied for general (n, k) fork-join systems, see, e.g., [Joshi et al., 2012; Shah et al., 2014; Wang et al., 2018], but unfortunately no exact analysis exists when $n \geq 3$.

The same can be said regarding the modeling of speculative execution. As the scheme is mostly applied to distributed computing settings, where each incoming request has a set of tasks that need to be completed before the request is done, most existing results consider heuristics for joint codesign with task scheduling [Zaharia et al., 2008; Ren et al., 2015; Xu and Lau, 2017; Joshi, 2018]. These approaches do not consider queueing at the individual servers. An exception is [Aktaş and Soljanin, 2019], where the authors study speculative execution across servers modeled using *limited processor sharing*, i.e., a PS queue with a maximum amount of requests behind an FCFS queue.

On the contrary, exact analysis exists in the case of cloning. It was first studied by [Gardner et al., 2015] with servers modeled as single M/M/1 queues. Other notable contributions concerning cloning to single-queue servers with exponential distributions include [Qiu et al., 2016; Gardner et al., 2017; Ayesta, 2019]. Qiu et al. [Qiu et al., 2016] compare the use of multiple single-queue server models with a central queue. Gardner et al. [Gardner et al., 2017] derived results on the largest marginal improvement that can be obtained using the *Redundancy-d* cloning policy, which clones each request to exactly d servers. Ayesta et al. [Ayesta, 2019] improved the analysis of Redundancy-d, including using CoS.

Subsequently, researchers began investigating cloning with specific probability distributions for interarrival times and service times, identifying the characteristics of the stochastic processes that make cloning beneficial [Shah et al., 2016]. In [Qiu et al., 2017], the results regarding the usage of a central queue were extended to the no-cancel policy, PH distributed service times and arrivals according to a Markovian arrival process. Furthermore, Redundancy-d over homogeneous servers with scaled Bernoulli-distributed service times was studied in [Raaijmakers et al., 2019]. In [Joshi et al., 2015], and later in [Joshi et al., 2017; Joshi, 2018], many of the results obtained with the M/M/1 model were extended to the M/G/1 model. However, an underlying assumption for the extension was that all service time distributions are independent and identically distributed (IID), which rules out heterogeneity.

In an attempt to bring 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. Cancellation overheads were also briefly touched upon in the model used in [Joshi, 2018].

A common theme with the above-cited works for cloning is that they all rely on the FCFS discipline, either for the central or the server-side queues.

In Chapter 6, we introduce the concept of *synchronized service* that enables certain server-side cloning systems to be equivalently represented as a single G/G/k queue. This extends the previous state-of-the-art, as the equivalent G/G/k queue requires no assumptions on either interarrival or service time distributions for the involved queues, effectively handling both dependencies and heterogeneity. It is also valid in any queueing discipline and any number of processors k, as long as they are the same across all queues. However, the considered cloning system must fulfill the assumptions for synchronized service, which are limiting and not very realistic to obtain in practice. Focusing on the PS discipline, we thus study what happens if these assumptions are relaxed. Finally, the concept is used to study server systems under speculative execution.

3

An Improved Fluid Model for PS Queueing Networks

The fluid model offers one potential way of quickly evaluating important performance metrics, both at stationary and transient times, for large-scale queueing networks that are not in product-form. However, it is in general hard to find a good fluid model of a queueing network, and most results consider single-queue models. If the queueing network can be reduced to a density-dependent population process, then the mean-field approximation yields one quick way of obtaining an often adequate fluid model.

Introduction

Recently, it was shown in [Pérez and Casale, 2013; Pérez and Casale, 2017] that closed multi-class networks of PS and INF queues, where each class is assumed to have a PH distributed service time, can be translated into a density-dependent population process that fulfills Kurtz's theorem. This is of high interest for performance modeling of cloud applications, as PS and INF queues are common when modeling cloud systems and that multi-class networks with PH service times can capture quite general system structures. In this chapter, we extend these two previous results by proving that mixed PS networks can also be translated into a density-dependent population process that fulfills Kurtz's theorem. We also introduce a compact matrix-form representation that describes the drift function in a mixed PS network. This drift function is shown to be globally Lipschitz, implying that the mean-field fluid model will always have a solution.

Moreover, despite its theoretical convergence properties, it is well-known that the mean-field approximation can experience serious errors when the system size is small. To improve the accuracy of the mean-field fluid model in these cases, we in this chapter also derive a simple technique based on data-driven smoothing of the drift function. The technique is computationally

cheap even at very large scales, in contrast to existing more general refined mean-field models such as [Gast and Van Houdt, 2017; Randone et al., 2021].

We also derive an accurate closed-form approximation of the entire response time CDF over any subset of classes and queues in the network. This contrasts with the current state of the art for mean-field fluid models, which achieves this by solving an extended state fluid model [Zhu et al., 2020].

The contributions are finally evaluated in a large simulation experiment, which shows that they can be used to accurately predict performance metrics of a simulated cloud application under the system perturbations common in cloud computing; Workload change, horizontal scaling and vertical scaling.

Outline. This chapter is structured as follows. In the end of this introduction, some necessary modeling assumptions and two example systems are introduced. In Section 3.1, we provide the necessary background on the mean-field fluid model for closed PS networks. In Section 3.2, we introduce the compact matrix form for the drift of mixed networks and prove its mean-field convergence. In Section 3.3, we present the mean-field improvements using smoothing, and following that, in Section 3.4, we give the closed-form approximation of the response time CDF. To validate our claims, an extensive simulation experiment is reported in Section 3.5. Finally, the chapter is summarized in Section 3.6.

Assumptions and Notation

We assume a queueing network where $\mathcal{Q}$ denotes its set of queues, and k_i the amount of servers in queue i. All queues in the network are assumed to follow either the PS or INF discipline. Furthermore, let $\mathcal{C}$ be the total set of classes in the queueing network, and $\mathcal{C}_i$ the set of classes in queue i. We assume that each class has its own service time distribution, and further that all classes are unique such that $|\mathcal{C}| = \sum_{i \in \mathcal{Q}} |\mathcal{C}_i|$. Each class can potentially be subject to Poisson arrivals as defined by the rates in $\boldsymbol{\lambda} \in \mathbb{R}_+^{|\mathcal{C}| \times 1}$.

To keep track of class-to-class transitions and departures in the network, we introduce the substochastic matrix $P \in \mathbb{R}_+^{|\mathcal{C}| \times |\mathcal{C}|}$, such that $\sum_{j,s} P_{i,j}^{r,s} \leq 1$ where $P_{i,j}^{r,s} \geq 0$ gives the probability that a request completed in class r in queue i gets routed to class s in queue j. We disallow completed requests from any class/queue to be directly routed back to itself, hence $P_{i,i}^{r,r} = 0 \; \forall (i,r) \in (\mathcal{Q}, \mathcal{C}_i)$. Let the dot notation in the sub/superscript denote a submatrix over all the affected elements, e.g., $P_{i,j}^{\cdot,\cdot}$ represents the matrix in $\mathbb{R}_+^{|\mathcal{C}_i| \times |\mathcal{C}_j|}$ of class-to-class transition probabilities from $i \in \mathcal{Q}$ to $j \in \mathcal{Q}$.

The PH distribution and phase states. Each class in each queue in the network is assumed to have a PH distributed service time. This implies that we for each $(i,r) \in (\mathcal{Q}, \mathcal{C}_i)$ will have an internal, single-sink Markov chain. Denote its set of transient states as $\mathcal{S}_{i,r}$, and the state-to-state transition

rates as $\Psi^{i,r} \in \mathbb{R}^{|\mathcal{S}_{i,r}|\times|\mathcal{S}_{i,r}|}$ between the transient states, and $\psi^{i,r} \in \mathbb{R}_+^{|\mathcal{S}_{i,r}|\times 1}$ from the transient states to the single absorbing state. Due to properties of the CTMC, $\Psi^{i,r}_{a,a} \leq 0$, $\Psi^{i,r}_{a,b} \geq 0$, and $\sum_b \Psi^{i,r}_{a,b} + \psi^{i,r}_a = 0$ for any $a, b \in \mathcal{S}_{i,r}$. Furthermore, let $\zeta^{i,r} \in \mathbb{R}_+^{|\mathcal{S}_{i,r}|\times 1}$ be the entrance probability vector of the same PH distribution.

A request entering (i, r) will be assigned to its internal states according to $\zeta^{i,r}$. Once it reaches the absorbing state, it will immediately depart for another class in the network according to $P^{r,\cdot}_{i,\cdot}$, or depart the network entirely. This implies that the location of all requests in the network is captured by the transient states alone at any time. We will refer to these transient states as *phase states* or simply *states*.

Let $\mathcal{S}$ denote the total set of phase states in the network, and let the vector valued stochastic variable $\boldsymbol{X}(t) \in \mathbb{Z}_+^{|\mathcal{S}|\times 1}$ be the number of requests populating the phase states at a given time instance $t \in \mathbb{R}_+$. Although the states $\boldsymbol{X}$ are time dependent, we will for brevity not be consistent in writing it out. Let $\boldsymbol{X}^*$ denote this stochastic variable at stationarity. Following the notation of [Pérez and Casale, 2017], let $X_{i,r,a}$ denote the population of requests in queue i, class $r \in \mathcal{C}_i$, and phase state $a \in \mathcal{S}_{i,r}$. To map a state $a \in \mathcal{S}$ to its corresponding queue, we introduce $Q(a) \in \mathcal{S} \to \mathcal{Q}$, and we let $\boldsymbol{X}_{Q(a)} \subset \boldsymbol{X}$ be the vector of all states in the same queue as state a.

The concept of chains. Similarly to [Bolch et al., 2006], we define the concept of chains as follows.

Definition 3.1
The set of classes $\mathcal{C}$ can be partitioned into a set of chains Γ, such that there exists no path in the network from any class in a chain to any class in any other chain. Each chain $\gamma \in \Gamma$ is constructed from collections of $(i, r) \in (\mathcal{Q}, \mathcal{C}_i)$, and every class will belong to exactly one chain.

Let γ_o denote an open chain, and γ_c a closed chain. Due to the existence of arrivals and departures in open chains, it holds that $\exists(i, r) \in \gamma_o$ s.t. $\lambda^{i,r} > 0$ and $\sum_{j,s} P^{r,s}_{i,j} < 1$, while for closed chains $\sum_{j,s} P^{r,s}_{i,j} = 1, \lambda^{i,r} = 0\ \forall(i, r) \in \gamma_c$. This implies that the total population for each closed chain is constant over time, while it can vary for open chains. Also, let $\Gamma_o \subset \Gamma$ and $\Gamma_c \subset \Gamma$ be the sets of open and closed chains in a network. Further, we assume the following.

Assumption 3.1
Every open chain γ_o is *transient*, and that every closed chain γ_c is *irreducible*.

A transient chain implies that a request starting from any $(i, r) \in \gamma_o$ will leave the network in finite time, while an irreducible chain implies that there is a nonzero probability that, starting from any $(i, r) \in \gamma_c$, we will visit any $(j, s) \in \gamma_c$ in a finite amount of steps.

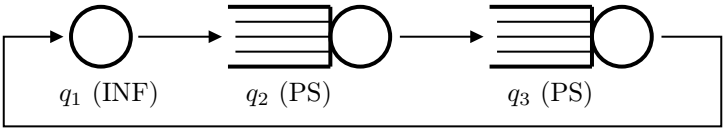


Figure 3.1 The closed cyclic network for the second running example.

Two Simple Running Examples

To provide some intuition regarding the new theory and methods to be presented, here we introduce two simple but illustrative examples that will follow us in the remainder of this chapter.

Example 1. As a first example, we consider one of the simplest open queueing networks possible, namely, the M/M/1-PS queue. It follows the PS discipline, has a single server, and a single class with an exponentially distributed service rate subjected to external Poisson arrivals. Two parameters are free to set, namely λ—the rate of the Poisson arrival process and μ—the rate of the exponential service time distribution. We set $\mu = 1$ and let λ vary in the experiments.

Example 2. As a slightly more advanced example, we consider a closed cyclic queueing network of three single-class queues where queue 1 (q_1) is an INF queue, and queues 2 and 3 (q_2, q_3) are PS queues. The network can be seen in Figure 3.1. This closed network is given a population of $N = 50$, and the two PS queues q_2, q_3 are given $k_2 = 4, k_3 = 8$ servers. Further, q_1 is given an exponential service time with rate μ_1 which we will let vary in the experiments, while q_2 and q_3 are given Coxian service time distributions with rates $\mu_2 = 0.5, \mu_3 = 1$ and the coefficient of variations $\sigma_2\mu_2 = 0.5, \sigma_3\mu_3 = 10$, resulting in the PH distribution matrices

$$\Psi^{2,1} = \begin{bmatrix} -4.0 & 4.0 \\ 0 & -4.0 \end{bmatrix}, \quad \psi^{2,1} = \begin{bmatrix} 0 & 4.0 \end{bmatrix},$$
$$\Psi^{3,1} = \begin{bmatrix} -2.0 & 0.1 \\ 0 & -0.1 \end{bmatrix}, \quad \psi^{3,1} = \begin{bmatrix} 1.9 & 0.1 \end{bmatrix}, \tag{3.1}$$

and where $\zeta^{2,1} = \zeta^{3,1} = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

3.1 The Mean-Field Fluid Model for Closed PS Networks

In [Pérez and Casale, 2013; Pérez and Casale, 2017], it is shown that a *closed* network of multi-class PS and INF queues with PH distributed service times can be expressed as a type of CTMC known as a density-dependent population process. The possible transitions in the Markov chain are given by either

a request moving from a transient phase $a \in \mathcal{S}_{i,r}$ to another $b \in \mathcal{S}_{i,r}$ in some $(i,r) \in (\mathcal{Q}, \mathcal{C}_i)$, or by the request leaving the phases of (i,r) and entering the phase $b \in \mathcal{S}_{j,s}$ in $(j,s) \in (\mathcal{Q}, \mathcal{C}_j)$. Let $\boldsymbol{e}_i \in \mathbb{Z}_+^{|\mathcal{S}|\times 1}$ denote a zero vector with a single value of 1 at index i, and let $f(\boldsymbol{X}, l) \in \mathbb{R}_+^{|S|\times 1} \times \mathbb{Z}_+^{|S|\times 1} \to \mathbb{R}_+$ describe a function for the density-dependent transition rate for some transition l. As presented in [Pérez and Casale, 2013; Pérez and Casale, 2017], the possible transition types and transition rate functions are given by:

Between classes: transition type $l^c = \boldsymbol{e}_{j,s,b} - \boldsymbol{e}_{i,r,a}$ and

$$f^c(\boldsymbol{X}, \boldsymbol{e}_{j,s,b} - \boldsymbol{e}_{i,r,a}) = \psi_a^{i,r} \zeta_b^{j,s} P_{i,j}^{r,s} \theta_{i,r,a}(\boldsymbol{X}). \tag{3.2}$$

Between phases: transition type $l^n = \boldsymbol{e}_{i,r,b} - \boldsymbol{e}_{i,r,a}$ and

$$f^n(\boldsymbol{X}, \boldsymbol{e}_{i,r,b} - \boldsymbol{e}_{i,r,a}) = \Psi_{a,b}^{i,r} \theta_{i,r,a}(\boldsymbol{X}), \tag{3.3}$$

where $\theta_{i,r,a}(\boldsymbol{X}) = X_{i,r,a} g_{i,r,a}(\boldsymbol{X})$. We will refer to the function $g(\boldsymbol{X}) \in \mathbb{R}_+^{|S|\times 1} \to [0,1]^{|S|\times 1}$ as the *processor share* of $\boldsymbol{X}$ which is given as

$$g_{i,r,a}(\boldsymbol{X}) = \frac{\min\left(k_i,\ \sum_{s\in\mathcal{C}_i} \sum_{b\in\mathcal{S}_{i,s}} X_{i,s,b}\right)}{\sum_{s\in\mathcal{C}_i} \sum_{b\in\mathcal{S}_{i,s}} X_{i,s,b}}. \tag{3.4}$$

Let $\mathcal{L}$ be the set of all possible transitions in the CTMC. The drift of the population process is then given by

$$F(\boldsymbol{X}) = \sum_{l\in\mathcal{L}} l f(\boldsymbol{X}, l). \tag{3.5}$$

The CTMC dynamics can be evaluated using simulations, but this gives rise to a high computational cost as the state space grows. Instead, using a mean-field approximation it is possible to approximate $\mathbb{E}(\boldsymbol{X}(t))$ with the solution of a system of ODEs. Using (3.5), we can introduce $\boldsymbol{x}(t) \in \mathbb{R}_+^{|\mathcal{S}|\times 1}$ as a continuous-time vector of states that is given as the solution to

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= F(\boldsymbol{x}(t)), \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0). \end{aligned} \tag{3.6}$$

If we introduce a sequence of population processes, $\left\{\boldsymbol{X}^{(v)}\right\}_{v\geq 1}$, such that for each v, the population process jumps from $\boldsymbol{X}^{(v)}$ to $\boldsymbol{X}^{(v)} + l$ at a rate of $vf\left(\frac{1}{v}\boldsymbol{X}^{(v)}, l\right)$ for each $l \in \mathcal{L}$, it can be shown under some mild assumptions [Kurtz, 1970, Th.3.1] that for every $\delta > 0$,

$$\lim_{v\to+\infty} \mathbb{P}\left\{\sup_{t\leq T}\left|\frac{\boldsymbol{X}^{(v)}(t)}{v} - \boldsymbol{x}(t)\right| > \delta\right\} = 0. \tag{3.7}$$

I.e., the error of the mean-field approximation goes to zero as the system size v approaches infinity. As shown in [Pérez and Casale, 2017, Th.1] a sequence of closed queueing networks of PS and INF queues, where both the initial population N and servers per queue k_i are scaled as vN and vk_i, fulfills this convergence property.

3.2 Extension to Mixed PS Networks

The convergence property according to Kurtz's theorem has so far only been shown to hold for closed networks of PS and INF queues. In this chapter, we will extend this result and show that it holds for *mixed* multi-class networks as well. First, we must introduce the concepts of external arrivals and departures to the queueing network in a form that is compatible with the transformation to a density-dependent population process.

For the external arrivals, we let each class in $\mathcal{C}$ be subjected to Poisson arrivals with the rates defined in $\boldsymbol{\lambda}$. This yields a new type of transition $l^{\lambda} = \boldsymbol{e}_{i,r,a}$ in the population process, where a request arrives at some phase state $(i, r, a) \in (\mathcal{Q}, \mathcal{C}_i, \mathcal{S}_{i,r})$ with the transition rate function

$$f^{\lambda}(\boldsymbol{X}, \boldsymbol{e}_{i,r,a}) = \zeta_a^{i,r} \lambda^{i,r}. \tag{3.8}$$

By definition, $\lambda^{i,r} = 0 \ \forall (i,r) \in \Gamma_c$. Furthermore, we let requests leaving $(i,r) \in (\mathcal{Q}, \mathcal{C}_i)$ have the probability $1 - \sum_{j,s} P_{i,j}^{r,s}$ to depart the network entirely. Considering some phase $a \in \mathcal{S}_{i,r}$, this yields a new transition type $l^d = -\boldsymbol{e}_{i,r,a}$ with the transition rate function

$$f^d(\boldsymbol{X}, -\boldsymbol{e}_{i,r,a}) = \psi_a^{i,r} \left(1 - \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P_{i,j}^{r,s} \right) \theta_{i,r,a}(\boldsymbol{X}). \tag{3.9}$$

Since the drift function (3.5) is defined as the sum over all possible transitions times their rate functions, we can then write out the drift affecting an arbitrary state i, r, a as follows:

$$\begin{aligned} F_{i,r,a}(\boldsymbol{X}) = &- \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} \sum_{b \in \mathcal{S}_{j,s}} \psi_a^{i,r} \zeta_b^{j,s} P_{i,j}^{r,s} \theta_{i,r,a}(\boldsymbol{X}) \\ &+ \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} \sum_{b \in \mathcal{S}_{j,s}} \psi_b^{j,s} \zeta_a^{i,r} P_{j,i}^{s,r} \theta_{j,s,b}(\boldsymbol{X}) \\ &- \sum_{b \in \mathcal{S}_{i,r}}^{b \neq a} \Psi_{a,b}^{i,r} \theta_{i,r,a}(\boldsymbol{X}) + \sum_{b \in \mathcal{S}_{i,r}}^{b \neq a} \Psi_{b,a}^{i,r} \theta_{i,r,b}(\boldsymbol{X}) \\ &- \psi_a^{i,r} \left(1 - \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P_{i,j}^{r,s} \right) \theta_{i,r,a}(\boldsymbol{X}) + \zeta_a^{i,r} \lambda^{i,r}. \end{aligned} \tag{3.10}$$

Factorizing all $\theta_{i,r,a}(\boldsymbol{X})$ and using $\sum_b \Psi_{a,b} + \psi_a^{i,r} = 0$ yields that

$$F_{i,r,a}(\boldsymbol{X}) = \sum_{b \in \mathcal{S}_{i,r}} \Psi_{b,a}^{i,r} \theta_{i,r,b}(\boldsymbol{X}) + \zeta_a^{i,r} \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} \sum_{b \in \mathcal{S}_{j,s}} \psi_b^{j,s} P_{j,i}^{s,r} \theta_{j,s,b}(\boldsymbol{X}) + \zeta_a^{i,r} \lambda^{i,r}. \tag{3.11}$$

Compact Matrix-Form Representation of the Drift Function

To represent the drift function in a more manageable form, we will here derive a simple matrix expression for it, which is easily obtainable given the routing probability matrix P, the arrival rates $\boldsymbol{\lambda}$ and the matrices of the PH distributions $\left(\Psi^{i,r}, \psi^{i,r}, \zeta^{i,r}\right)$ $\forall (i,r) \in (\mathcal{Q}, \mathcal{C}_i)$. Thus far, we have not assumed any particular ordering of the states, but this is now needed.

Assumption 3.2
Let $X_1 = X_{1,1,1}$. For every i, r, a, s.t. $X_j = X_{i,r,a}$, define for $j+1 \leq |\mathcal{S}|$

$$X_{j+1} = \begin{cases} X_{i,r,a+1} & \text{if} \quad a+1 \leq |\mathcal{S}_{i,r}|, \\ X_{i,r+1,1} & \text{if} \quad a+1 > |\mathcal{S}_{i,r}|, \ r+1 \leq |\mathcal{C}_i|, \\ X_{i+1,1,1} & \text{if} \quad a+1 > |\mathcal{S}_{i,r}|, \ r+1 > |\mathcal{C}_i|. \end{cases} \tag{3.12}$$

I.e., the phases of subsequent classes are subsequently ordered, and the queues containing those classes are in turn also subsequently ordered.

Given this ordering, the positions of the states $\mathcal{S}_{i,r}$ for $(i,r) \in (\mathcal{Q}, \mathcal{C}_i)$ in $\mathcal{S}$ are given by $\vec{m}_{i,r} = m_{i,r} : m_{i,r} + |\mathcal{S}_{i,r}|$, where $m_{i,r} = \sum_{i_m=1}^{i_m<i} \sum_{r_m=1}^{|C_{im}|} |\mathcal{S}_{i_m,r_m}| + \sum_{r_m=1}^{r_m<r} |\mathcal{S}_{i,r_m}|$. Similarly, the position of the classes $\mathcal{C}_i$ for $i \in \mathcal{Q}$ in $\mathcal{C}$ is given by $\vec{n}_i = n_i : n_i + |C_i|$ where $n_i = \sum_{i_n=1}^{i_n<i} |\mathcal{C}_{i_n}|$. Also, the position of $(i,r) \in (\mathcal{Q}, \mathcal{C}_i)$ in $\mathcal{C}$ is given by $n_{i,r} = n_i + r$. The following theorem can then be stated.

Theorem 3.1—Compact Matrix-Form
Given Assumption 3.2, the drift function of any mixed queueing network of PS and INF queues can be expressed as

$$F(\boldsymbol{X}) = \boldsymbol{W}^T \theta(\boldsymbol{X}) + \boldsymbol{A}\boldsymbol{\lambda}, \tag{3.13}$$

where

- $\theta(\boldsymbol{X}) \in \mathbb{R}_+^{|\mathcal{S}| \times 1} \to \mathbb{R}_+^{|\mathcal{S}| \times 1}$ s.t.

$$\theta_a(\boldsymbol{X}) = X_a \frac{\min\left(k_{Q(a)}, \sum \boldsymbol{X}_{Q(a)}\right)}{\sum \boldsymbol{X}_{Q(a)}} \quad \forall a \in \mathcal{S},$$

- $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$,
- $\boldsymbol{P} \in \mathbb{R}_+^{|\mathcal{C}|\times|\mathcal{C}|}$ s.t. $\boldsymbol{P}_{\vec{n}_i,\vec{n}_j} = P^{\cdot,\cdot}_{i,j} \quad \forall i,j \in \mathcal{Q}$,
- $\boldsymbol{\Psi} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$, $\boldsymbol{B} \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{C}|}$ and $\boldsymbol{A} \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{C}|}$ are block diagonal matrices s.t $\forall (i,r) \in (\mathcal{Q},\mathcal{C}_i)$,

$$\boldsymbol{\Psi}_{\vec{m}_{i,r}\vec{m}_{i,r}} = \Psi^{i,r},\ \boldsymbol{B}_{\vec{m}_{i,r},n_{i,r}} = \psi^{i,r},\ \boldsymbol{A}_{\vec{m}_{i,r},n_{i,r}} = \zeta^{i,r}.$$

Proof. See Appendix A. □

The theorem states that the entire mean-field fluid model can be decomposed into a product between (i) a constant matrix $\boldsymbol{W}$ that describes the complete graph of transition rates between phase states, (ii) a vector valued function $\theta(\boldsymbol{x})$ that describes the effect of the queueing disciplines on those transition rates, and (iii) a vector $\boldsymbol{A}\boldsymbol{\lambda}$ that describes the external arrival rate to each phase state. At a glance it might look complex, but $\boldsymbol{W}$ can be created by simply combining block diagonals of the PH distribution matrices with $\boldsymbol{P}$.

This matrix-form can also be used to incorporate the method of approximating open chains with bottleneck queues [Bolch et al., 2006, p. 507]. After introducing the bottleneck queues, it is sufficient to set $\boldsymbol{\lambda} = \boldsymbol{0}$ and expand $\boldsymbol{P}$ and the block diagonal matrices $\boldsymbol{\Psi}, \boldsymbol{B}, \boldsymbol{A}$ to also cover these. If we then let $\boldsymbol{X}(0) \to +\infty$ in these new bottleneck queues, (3.2) would essentially turn into (3.8) for transitions between the bottleneck queues and their recipients, and the resulting mean-field fluid model should in general yield the same result as for the original model.

A comment on sparseness. As the size of $\boldsymbol{W}$ is $|\mathcal{S}| \times |\mathcal{S}|$, we can expect the number of possible elements in $\boldsymbol{W}$ to be large for interesting systems. However, the following can be stated regarding the sparsity of $\boldsymbol{W}$.

Proposition 3.1
$\boldsymbol{W}$ will be at least as sparse as the combined sparsity of $\boldsymbol{\Psi}, \boldsymbol{B}, \boldsymbol{P}, \boldsymbol{A}$, if

$$\max_{(i,r),(j,s)} \left(n_{\psi^{i,r}} \cdot n_{\zeta^{j,s}}\right) n_{\boldsymbol{P}} \leq n_{\boldsymbol{P}} + \sum_{i,r} \left(n_{\psi^{i,r}} + n_{\zeta^{i,r}}\right), \tag{3.14}$$

where n_M denotes the number of nonzero elements in some matrix M.

Proof. As

$$\boldsymbol{W}_{x,y} = \begin{cases} \Psi^{i,r}_{b,a} & \text{if } x = (i,r,a),\ y = (i,r,b), \\ \psi^{i,r}_a P^{r,s}_{i,j} \zeta^{j,s}_b & \text{if } x = (i,r,a),\ y = (j,s,b), \end{cases} \tag{3.15}$$

we get that

$$\begin{aligned} n_{\boldsymbol{W}} &= n_{\boldsymbol{\Psi}} + \sum_{i,r,a} \sum_{j,s,b} \mathbb{1}\left(\psi_a^{i,r} P_{i,j}^{r,s} \zeta_b^{j,s} > 0\right) \\ &= n_{\boldsymbol{\Psi}} + \sum_{i,r} \sum_{j,s} n_{\psi^{i,r}} n_{\zeta^{j,s}} \mathbb{1}\left(P_{i,j}^{r,s} > 0\right) \\ &\leq n_{\boldsymbol{\Psi}} + \max_{(i,r),(j,s)} \left(n_{\psi^{i,r}} n_{\zeta^{j,s}}\right) n_{\boldsymbol{P}}. \end{aligned} \tag{3.16}$$

As the total sparsity of $\boldsymbol{\Psi}, \boldsymbol{B}, \boldsymbol{P}, \boldsymbol{A}$ is $n_{\boldsymbol{\Psi}} + n_{\boldsymbol{P}} + \sum_{i,r} n_{\psi^{i,r}} + n_{\zeta^{i,r}}$, the inequality is shown. □

Sparseness is a desirable property, as it reduces the number of computations needed when evaluating a matrix expression. Using, e.g., the common Coxian PH distribution [Cox, 1955], where $\zeta_1 = 1$, $\zeta_a = 0\ \forall a > 1$ and where each internal transient state n can only transmit to $n+1$ or the absorbing state, would yield both a small $\max_{(i,r),(j,s)} \left(n_{\psi^{i,r}} n_{\zeta^{j,s}}\right)$ and a sparse Ψ. Further, we would in general expect that request paths are fairly deterministic for queueing network models used in cloud computing, leading to a sparse $\boldsymbol{P}$. These paths ultimately model the often fixed order that resources are visited by external requests to complete their service. Requests can however be load balanced over multiple instances of the same resource which decreases sparsity of $\boldsymbol{P}$, but only marginally if the number of different resources are large. Hence, from Proposition 3.1 we could for many cases expect that $\boldsymbol{W}$ is both sparse and that we would actually gain sparseness by creating it.

The Resulting Mean-Field Fluid Model

The drift function introduced in Theorem 3.1 yields the following mean-field fluid model for mixed networks of PS and INF queues,

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= \boldsymbol{W}^T \theta(\boldsymbol{x}) + \boldsymbol{A}\boldsymbol{\lambda}, \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0). \end{aligned} \tag{3.17}$$

We refer the point $\boldsymbol{x}^*$ as a *fixed point* iff $F(\boldsymbol{x}^*) = \boldsymbol{W}^T \theta\left(\boldsymbol{x}^*\right) + \boldsymbol{A}\boldsymbol{\lambda} = 0$.

Further, the drift function obtains the following important property.

THEOREM 3.2
The drift function of the mixed network of INF and PS queues (3.13) is Lipschitz continuous in $\mathbb{R}_+^{|\mathcal{S}|\times 1}$.

Proof. See Appendix A. □

Via Picard-Lindelöf's theorem [Sideris, 2013, Section 3.3], this global Lipschitz property implies that the corresponding mean-field fluid model (3.17) is guaranteed to have an unique solution for each choice of $\boldsymbol{x}(0) \in \mathbb{R}_+^{|S|\times 1}$.

Proof of mean-field convergence. We will now prove that there exists a sequence of mixed PS queueing networks under increasing system size that converges to the mean-field fluid model (3.17). The proof is similar to [Pérez and Casale, 2017, Th. 1] for the mean-field convergence of closed networks, but it cannot be directly applied as it relies on the property that closed networks have a fixed population size.

Introduce a sequence of mixed PS networks $\{\boldsymbol{X}^{(v)}\}_{v\geq 1}$, such that $\boldsymbol{X}^{(v)}(0) = v\boldsymbol{X}(0)$, $k_i^{(v)} = vk_i\ \forall i \in \mathcal{Q}$, and $\boldsymbol{\lambda}^{(v)} = v\boldsymbol{\lambda}$. The transition rate function $f^{(v)}$ for the arrivals can trivially be seen to uphold $f^{(v)}\left(\boldsymbol{X}^{(v)}, l\right) = vf\left(\frac{1}{v}\boldsymbol{X}^{(v)}, l\right)$. The remaining $f^{(v)}$ are all defined as some constant multiplied by $\theta_a^{(v)}(\boldsymbol{X})$ $a \in \mathcal{S}$, which becomes

$$\begin{aligned}\theta_a^{(v)}\left(\boldsymbol{X}^{(v)}\right) &= X_a^{(v)}\frac{\min\left(k_{Q(a)}^{(v)}, \sum \boldsymbol{X}_{Q(a)}^{(v)}\right)}{\sum \boldsymbol{X}_{Q(a)}^{(v)}} \\ &= \frac{1}{v}X_a^{(v)}\frac{\min\left(vk_{Q(a)}, \sum \frac{v}{v}\boldsymbol{X}_{Q(a)}^{(v)}\right)}{\sum \frac{1}{v}\boldsymbol{X}_{Q(a)}^{(v)}} = v\theta_a\left(\frac{1}{v}\boldsymbol{X}^{(v)}\right). \end{aligned} \tag{3.18}$$

Hence, $f^{(v)}\left(\boldsymbol{X}^{(v)}, l\right) = vf\left(\frac{1}{v}\boldsymbol{X}^{(v)}, l\right)\ \forall l \in \mathcal{L}$. The following theorem can then be stated.

THEOREM 3.3
Let $\{\boldsymbol{X}^{(v)}\}_{v\leq 1}$ be the sequence of mixed PS networks introduced above and $\boldsymbol{x}(t)$ the solution of the mean-field fluid model (3.17). Furthermore, let $\mathcal{E} \subset \mathbb{R}_+^{|\mathcal{S}|\times 1}$ be some open subset and $T \geq 0$ some finite constant, such that $\boldsymbol{x}(t) \in \mathcal{E}$, $\forall 0 \leq t \leq T$. Then, for any $\delta > 0$,

$$\lim_{v\to+\infty} \mathbb{P}\left\{\sup_{t\leq T}\left|\frac{\boldsymbol{X}^{(v)}(t)}{v} - \boldsymbol{x}(t)\right| > \delta\right\} = 0. \tag{3.19}$$

Proof. According to [Kurtz, 1970, Th. 3.1], this is fulfilled if

(i) $\lim_{v\to+\infty} v^{-1}\boldsymbol{X}^{(v)}(0) = \boldsymbol{x}(0)$,

(ii) for some constant $M_{\mathcal{E}}$, $||F(\boldsymbol{x}) - F(\boldsymbol{y})||_1 < M_{\mathcal{E}}||\boldsymbol{x} - \boldsymbol{y}||_1 \qquad \forall x, y \in \mathcal{E}$,

(iii) $\sup_{\boldsymbol{x}\in\mathcal{E}} \sum_{l\in\mathcal{L}} ||l||_1 f(\boldsymbol{x}, l) < +\infty$,

(iv) $\lim_{d\to+\infty} \sup_{\boldsymbol{x}\in\mathcal{E}} \sum_{||l||_1>d} ||l||_1 f(\boldsymbol{x}, l) = 0$.

(i) follows directly from $\boldsymbol{X}^{(v)}(0) = v\boldsymbol{X}(0)$, (ii) directly from Theorem 3.2, and (iv) directly from the fact that $\sup_{l\in\mathcal{L}} ||l||_1 = 2$. The remaining condition (iii) holds as all constants in any $f(\boldsymbol{x}, l)$ are finite, and $\theta_a(\boldsymbol{x})$ is bounded from above by $k_{Q(a)}$ for PS queues and by x_a for INF queues $\forall a \in \mathcal{S}$. □

This theorem does not depend on any assumption of the structure of $\boldsymbol{W}$ or $\boldsymbol{A\lambda}$ other than constant and finite elements. Hence, the convergence holds for any valid structure of $\boldsymbol{W}$ according to Theorem 3.1, such as multiple chains or even unstable queues. It can further be shown that if the mean-field fluid model converges to an unique fixed point, then the probability distribution of its corresponding sequence of population process $\boldsymbol{X}^{(v)}(t)$ converges to $\boldsymbol{x}^*$ no matter the order of the limits in t or v [Benaïm and Le Boudec, 2008].

A comment on global asymptotic stability. The property of convergence to an unique fixed point, $\lim_{t\to\infty} \boldsymbol{x}(t) = \boldsymbol{x}^*$, independent of the initial conditions $\boldsymbol{x}(0)$, is known as global asymptotic stability (GAS) and an often desirable trait in ODEs. Unfortunately, this is in general hard to prove for nonlinear systems such as (3.17). Regarding the mean-field approximation, some special cases where GAS can be found are discussed in [Gast, 2020]. We however suspect that for a general mixed PS network, then (3.17), if stable, converges to an unique fixed point independent of $\boldsymbol{x}(0)$ as long as the sum over each closed chain is fixed. We have no proof of this, but in accordance with [Zhu et al., 2020], we have found no system where it does not hold.

In certain special cases, this property becomes straight forward to prove. For example, if we consider a system of only INF queues.

Proposition 3.2
Given a mixed queueing network of only INF queues. Then (3.17) converges to an unique fixed point $\boldsymbol{x}^*$ for all $\boldsymbol{x}(0) \in \mathbb{R}_+$ where $\sum_{(i,r)\in\gamma_c} \sum_{a\in\mathcal{S}_{i,r}} x_{i,r,a}(0) = N_{\gamma_c}$ for each $\gamma_c \in \Gamma_c$. Here, N_{γ_c} is the population of the closed chain γ_c.

Before proving this, we will remark on some properties of the eigenvalues of $\boldsymbol{W}$. Let $\{\boldsymbol{W}_\gamma\}_{\gamma\in\Gamma}$ be a set of submatrices corresponding to the constant transition rates between each state in chain γ. Notice that, since each state belongs to exactly one class and Γ is a partition of $\mathcal{C}$, $\{\boldsymbol{W}_\gamma\}_{\gamma\in\Gamma}$ is a partition of $\boldsymbol{W}$, and subsequently $\boldsymbol{W}_{x,y} = 0\ \forall (x,y) \in \left(\mathcal{S}^{i,r}, \mathcal{S}^{j,s}\right)$ where $(i,r) \in \gamma_a$ and $(j,s) \in \gamma_b$ when $a \neq b$. The values and row sums of $\boldsymbol{W}_\gamma$, and thus $\boldsymbol{W}$, can be quantified in the same manner as $\boldsymbol{P}$ as follows.

Proposition 3.3
Let $V = \boldsymbol{W}_\gamma$, then $V_{i,i} \leq 0\ \forall i$, $V_{i,j} \geq 0\ \forall i \neq j$. Furthermore,

(i) If γ is closed, then $\sum_j V_{i,j} = 0\ \forall i$.

(ii) If γ is open, then $\sum_j V_{i,j} \leq 0\ \forall i$ and $\exists i\ s.t.\ \sum_j V_{i,j} < 0$.

Proof. The first statements, $V_{i,i} \leq 0\ \forall i$, $V_{i,j} \geq 0\ \forall i \neq j$ hold by construction as all involved elements are nonnegative except for the diagonal of $\boldsymbol{\Psi}$, for which the diagonal of V is a subset of.

For an arbitrary $a \in \mathcal{S}_{i,r}$, $(i,r) \in \gamma$, from (3.15) we get that $\sum_y V_{(i,r,a),y} = \sum_{b \in \mathcal{S}_{i,r}} \Psi^{i,r}_{a,b} + \psi^{i,r}_a \sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P^{r,s}_{i,j}$. Here, we have used that $\sum_b \zeta^{j,s}_b = 1$. As for PH distributions $\sum_{b \in \mathcal{S}_{i,r}} \Psi^{i,r}_{a,b} + \psi^{i,r}_a = 0$, (i) follows from that for closed chains $\sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P^{r,s}_{i,j} = 1$. (ii) follows from that for open chains that $\sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P^{r,s}_{i,j} \leq 1$ for all i, r and further as we by definition have departures $\exists (i,r)$ *s.t.* $\sum_{j \in \mathcal{Q}} \sum_{s \in \mathcal{C}_j} P^{r,s}_{i,j} < 1$. □

From this we can derive that all the eigenvalues $\mathcal{V}$ of $\boldsymbol{W}_\gamma$ have nonpositive real parts, which follows directly from Gershgorin's circle theorem [Varga, 2004]. Furthermore, for closed chains, we can recognize $\boldsymbol{W}_{\gamma_c}$ as a negative Laplacian matrix, and due to γ_c being irreducible, it will have exactly one zero-valued eigenvalue [Merris, 1994]. In a similar fashion, from [Shivakumar and Chew, 1974] we get that $\boldsymbol{W}_{\gamma_o}$ for an open chain is nonsingular, as γ_o is assumed to be transient. Nonsingularity implies that $\det(\boldsymbol{W}_{\gamma_o}) = \prod_{v \in \mathcal{V}} v \neq 0$; hence all of its eigenvalues have strictly negative real parts.

Proof of Proposition 3.2. For any state in an INF queue, we get that $\theta_{i,r,a}(\boldsymbol{x}) = x_{i,r,a}$. Thus the mean-field fluid model becomes a linear ODE on the form $\dot{\boldsymbol{x}} = \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{A}\boldsymbol{\lambda}$, where each chain can be considered separately.

For each open chain γ_o, $\boldsymbol{W}_{\gamma_o}$ only has eigenvalues of negative real part. The corresponding system is well-known to have a single steady-state solution independent of $\boldsymbol{x}(0)$.

For each closed chain γ_c, $\boldsymbol{W}_{\gamma_c}$ has one eigenvalue of value zero while the rest have negative real part. The steady-state solution of the corresponding system becomes a scaling of the eigenvector for the zero-valued eigenvalue [Sideris, 2013, Section 2.5]. As no flow can enter or leave the closed chain, this scaling must be the same for all $\boldsymbol{x}(0)$ in γ_c as long as its sum is N_{γ_c}.

Compact Matrix-Form of the Running Examples

Here we will show how the corresponding compact matrix-form of our two running examples introduced in the beginning of Chapter 3. According to Theorem 3.1, the compact matrix-form of the mean-field fluid model is given as $\frac{d\boldsymbol{x}}{dt} = \boldsymbol{W}^T \theta(\boldsymbol{x}) + \boldsymbol{A}\boldsymbol{\lambda}$ where $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$.

Example 1. For the M/M/1 queue, we get that $\boldsymbol{\Psi} = -1$, $\boldsymbol{B} = 1$, $\boldsymbol{A} = 1$ and $\boldsymbol{P} = 0$, resulting in $\boldsymbol{W} = -1$ and the mean-field fluid model

$$\frac{dx}{dt} = -\min(1, x) + \lambda. \tag{3.20}$$

Example 2. See Appendix B.

3.3 The Smoothed Mean-Field Fluid Model

It can be expected from Theorem 3.3 that, the more we scale $\boldsymbol{X}(0), k, \boldsymbol{\lambda}$, the more accurate the corresponding mean-field fluid model will be. However, it does not state how accurate the model is for a given system size, which is of interest when considering scenarios with nonideal (smaller) systems. This error is a known issue for general mean-field approximations, and it arises as

$$\frac{d}{dt}\mathbb{E}\left[\boldsymbol{X}\right] = \mathbb{E}\left[F\left(\boldsymbol{X}\right)\right] \neq F\left(\mathbb{E}\left[\boldsymbol{X}\right]\right) = \frac{d\boldsymbol{x}}{dt}, \tag{3.21}$$

in most cases. An exception is if the drift function is linear. It can further be shown that the approximation in fact corresponds to a first order approximation of $\mathbb{E}\left[F\left(\boldsymbol{X}\right)\right]$ [Bortolussi et al., 2013]. For the mixed PS queueing network, the mean drift is given as

$$\mathbb{E}\left[F(\boldsymbol{X})\right] = \boldsymbol{W}^T\mathbb{E}\left[\theta(\boldsymbol{X})\right] + \boldsymbol{A}\boldsymbol{\lambda}, \tag{3.22}$$

where it can be seen that the inequality arises if $\mathbb{E}\left[\theta\left(\boldsymbol{X}\right)\right] \neq \theta\left(\mathbb{E}\left[\boldsymbol{X}\right]\right)$. We can then compare the drift and its first order approximation by comparing the corresponding θ:

$$\begin{aligned}
\mathbb{E}\left[\theta_a(\boldsymbol{X})\right] &= \sum_{\boldsymbol{z}\in\mathbb{Z}_+^{|\mathcal{S}|\times 1}} \mathbb{P}\left(\boldsymbol{X}=\boldsymbol{z}\right) z_a \frac{\min\left(k_{Q(a)}, \sum \boldsymbol{z}_{Q(a)}\right)}{\sum \boldsymbol{z}_{Q(a)}}, \\
\theta_a\left(\mathbb{E}\left[\boldsymbol{X}\right]\right) &= \mathbb{E}\left[X_a\right] \frac{\min\left(k_{Q(a)}, \sum \mathbb{E}\left[\boldsymbol{X}_{Q(a)}\right]\right)}{\sum \mathbb{E}\left[\boldsymbol{X}_{Q(a)}\right]}.
\end{aligned} \tag{3.23}$$

As can be seen, the stochasticity of $\sum \boldsymbol{X}_{Q(a)}$ is not accounted for in the first order approximation, which can cause some peculiar errors close to $k_{Q(a)}$ when $\mathbb{E}\left[\sum \boldsymbol{X}_{Q(a)}\right]$ is either smaller or larger than $k_{Q(a)}$ while the PMF of $\boldsymbol{X}$ has a large support on the opposite side. As an illustration, consider the two running examples from Chapter 3.

Example 1:. The shortcomings of the mean-field fluid model become obvious when considering the M/M/1 queue. The fluid model becomes $\frac{dx}{dt} = -\min\left(1, x\right) + \lambda$ whose unique fixed point is given by $\boldsymbol{x}^* = \lambda$ as long as $\lambda \leq 1$. However, it is clear that the true mean $\mathbb{E}(X)$ of the Markov chain does not converge to this limit as its probability distribution is given by $\mathbb{P}\left(X = n\right) = \lambda^n\left(1-\lambda\right)$. In fact, $\mathbb{E}(X) = \lambda/\left(1-\lambda\right)$ can converge to any positive value if we let λ move arbitrarily close to 1, which makes the mean-field fluid model arbitrarily bad in this specific situation.

Example 2:. The errors observed above can also be found in closed networks. Example 2 is constructed such that both the PS queues have similar

mean service times but different support in the tails. If we vary μ_1 in an increasing sequence of $\mu_1 \in [0.05, 0.95]$ and for each μ_1 calculate $\boldsymbol{x}^*$ and simulate the system until stationarity to obtain $\hat{\boldsymbol{X}}^*$, we get the values seen in Figure 3.2. Here, it can clearly be seen that the mean-field fluid model fails to take into account the stochasticity of $\boldsymbol{X}$, and hence converges to a vastly different stationary solution for many values of μ_1.

Heuristic Utilization Tracking via P-Norm Smoothing

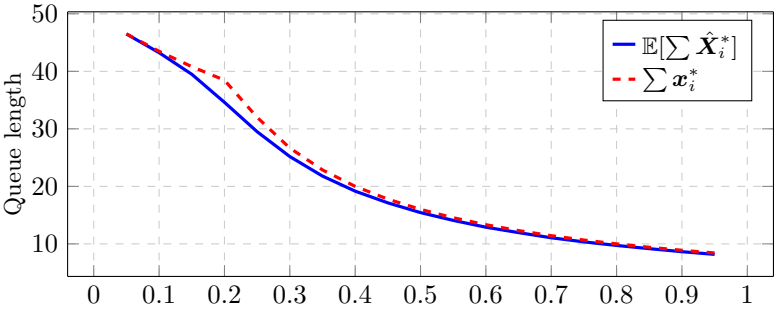
Instead of directly relying on the drift to obtain the time derivatives for the fluid model, one could try to form a better approximation to $\mathbb{E}[\theta(\boldsymbol{X})]$ by taking into account the stochasticity of $\boldsymbol{X}$. A natural way to achieve this is to include higher-order moments, as presented in [Gast and Van Houdt, 2017]. Unfortunately, this requires that $F(\boldsymbol{x})$ is at least twice differentiable, which is not true for the types of queueing networks we are considering, as $\theta_a(\boldsymbol{x})$, $a \in \mathcal{S}$, contains a min() function for each phase state in a queue that follows the PS discipline. To circumvent the differentiability requirement, another refined mean-field model was presented in [Randone et al., 2021] by solving multiple truncated versions of the problem. The accuracy of this refined model is dependent on the number of problem versions used, which makes for poor time scaling when accurate approximations for large scale systems are sought. Here, we will instead present a simple, novel approximation based on heuristic smoothing of the mean-field fluid model, which does not need any new introduction of states.

Using the processor share function, we can decompose $\theta_a(\boldsymbol{x}) = x_a g_a(\boldsymbol{x})$, where by construction $g_a(\boldsymbol{x}) = g_b(\boldsymbol{x}) \ \forall b \in \mathcal{S}_{Q(a),\cdot}$. Instead of forming an approximation with $\theta(\mathbb{E}[\boldsymbol{X}])$ directly, we can introduce a new function $h(\boldsymbol{x}, \boldsymbol{v}) \in \mathbb{R}_+^{|\mathcal{S}|\times 1} \times \mathbb{R}_+^{|\mathcal{S}|\times 1} \to \mathbb{R}_+^{|\mathcal{S}|\times 1}$ with some time dependent parameter vector $\boldsymbol{v}(t) \in \mathbb{R}^{|\mathcal{S}|\times 1}$, such that for any valid $\boldsymbol{X}$,

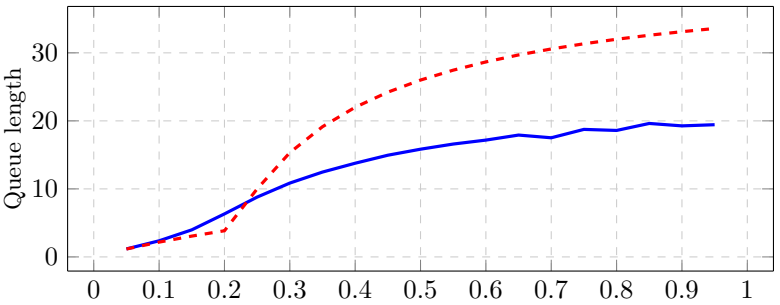
$$\mathbb{E}[X_a]\, h_a(\mathbb{E}[\boldsymbol{X}], \boldsymbol{v}) = \mathbb{E}[\theta_a(\boldsymbol{X})] \quad \forall a \in \mathcal{S}. \tag{3.24}$$

The parameter $\boldsymbol{v}$ is needed as $\mathbb{E}[\theta_a(\boldsymbol{X})]$ is dependent on the PMF of $\boldsymbol{X}$, and not only its mean. It is desirable for this function that $\boldsymbol{v}$ can be easily determined, and that it is robust in the sense that, for a given $\boldsymbol{v}$, small changes in $\boldsymbol{X}$ should still yield a good approximation in (3.24). The question is whether it is possible to find such a function. We can try to approximate it by replacing $h(\boldsymbol{x}, \boldsymbol{v})$ with an approximation $\hat{g}(\boldsymbol{x}, \boldsymbol{\eta})$ by introducing the processor share constraint $\hat{g}_a(\boldsymbol{x}, \boldsymbol{\eta}) = \hat{g}_b(\boldsymbol{x}, \boldsymbol{\eta}) \ \forall b \in \mathcal{S}_{Q(a),\cdot}$, where $\boldsymbol{\eta}(t) \in \mathbb{R}^{|\mathcal{Q}|\times 1}$. This constraint allows us to approximate the total $\mathbb{E}[\theta(\boldsymbol{X})]$ in each queue i as

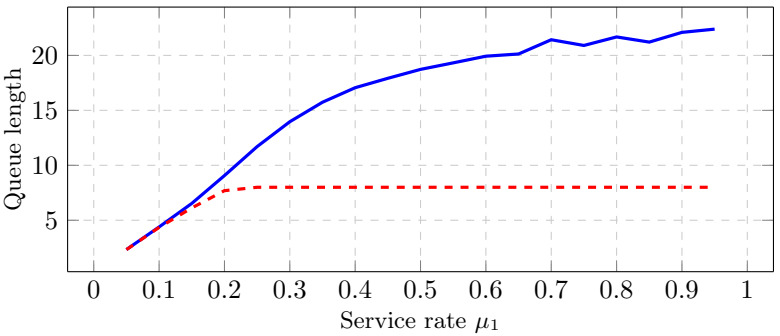
$$\sum \mathbb{E}[\boldsymbol{X}_i] \odot \hat{g}_i(\mathbb{E}[\boldsymbol{X}], \boldsymbol{\eta}) = \hat{g}_i(\mathbb{E}[\boldsymbol{X}], \boldsymbol{\eta}) \sum \mathbb{E}[\boldsymbol{X}_i]. \tag{3.25}$$



(a) Stationary mean queue length in INF queue $i = 1$.



(b) Stationary mean queue length in PS queue $i = 2$.



(c) Stationary mean queue length in PS queue $i = 3$.

Figure 3.2 Comparison of the stationary mean queue lengths in Example 2 over different service rates for queue 1, considering values obtained from simulation (blue line) and the mean-field fluid model (red dashed). As $\boldsymbol{X}_i$ where $i \in \mathcal{Q}$ gives the vector of requests in each state belonging to i, the total queue length is given as $\sum \boldsymbol{X}_i$.

Equivalently, summing over all states in the same queue as phase state a for the right hand side in (3.24) yields

$$\begin{aligned}\sum \mathbb{E}\left[\boldsymbol{X}_i \odot \boldsymbol{g}_i\left(\boldsymbol{X}\right)\right] &= \sum_{a \in \mathcal{S}_{i,\cdot}} \sum_{\boldsymbol{z} \in \mathbb{Z}_+^{|\mathcal{S}|\times 1}} \mathbb{P}\left(\boldsymbol{X}=\boldsymbol{z}\right) z_a \frac{\min\left(k_i, \sum \boldsymbol{z}_i\right)}{\sum \boldsymbol{z}_i} \\ &= \sum_{\boldsymbol{z} \in \mathbb{Z}_+^{|\mathcal{S}|\times 1}} \mathbb{P}\left(\boldsymbol{X}=\boldsymbol{z}\right) \min\left(k_i, \sum \boldsymbol{z}_i\right), \end{aligned} \tag{3.26}$$

which can be recognized as the total *utilization* over queue i, i.e., $\rho_i(\boldsymbol{X})$, multiplied by k_i [Bolch et al., 2006]. The utilization is usually considered in stationarity, but here we consider $\boldsymbol{X}$ at an arbitrary $t \geq 0$, hence the dependence.

By then setting (3.25) equal to (3.26) $\forall i \in \mathcal{Q}$, the following necessary condition for equality in (3.24) for $\hat{g}(\boldsymbol{x}, \boldsymbol{\eta})$ is obtained

$$\hat{g}_i\left(\mathbb{E}\left[\boldsymbol{X}\right], \boldsymbol{\eta}\right) = \frac{\sum_{c \geq 0} \mathbb{P}\left(\sum \boldsymbol{X}_i = c\right) \min\left(k_i, c\right)}{\sum \mathbb{E}\left[\boldsymbol{X}_i\right]} = \frac{k_i \rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]}. \tag{3.27}$$

The value of $\rho_i\left(\boldsymbol{X}\right)$ is dependent on the PMF of $\boldsymbol{X}$, but we can isolate the following two extreme cases for (3.27).

Case 1: $\mathbb{P}\left(\sum \boldsymbol{X}_i \leq k_i\right) = 1$,

$$\frac{k_i \rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} = \frac{\sum_{c \geq 0} \mathbb{P}\left(\sum \boldsymbol{X}_i = c\right) c}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} = 1. \tag{3.28}$$

Case 2: $\mathbb{P}\left(\sum \boldsymbol{X}_i \leq k_i\right) = 0$,

$$\frac{k_i \rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} = \frac{\sum_{c \geq 0} \mathbb{P}\left(\sum \boldsymbol{X}_i = c\right) k_i}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} = \frac{k_i}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]}. \tag{3.29}$$

Due to the minimum, (3.27) is always bounded from above by these two extreme cases. Furthermore, if we assume that there is no uncertainty in $\boldsymbol{X}$ for any $t \geq 0$, then one of the two extremes will always hold, and it will give us back the original $g_i\left(\mathbb{E}\left[\boldsymbol{X}\right]\right)$ and with it the mean-field fluid model. Further, (3.27) is bounded from below by 0. Hence, for any valid $\boldsymbol{X}$

$$0 \leq \frac{k_i \rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum X_i\right]} \leq \frac{\min\left(k_i, \mathbb{E}\left[\sum \boldsymbol{X}_i\right]\right)}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]}. \tag{3.30}$$

It is difficult to say how an arbitrary time evolving PMF of $\boldsymbol{X}_i$ in a queueing network behaves, but we can assume that $\mathbb{P}\left(\sum \boldsymbol{X}_i \leq k_i\right)$ is in general decreasing in $\mathbb{E}\left[\sum \boldsymbol{X}_i\right]$. This implies that we can expect $k_i \rho_i(\boldsymbol{X}) / \mathbb{E}\left[\sum X_i\right]$ to

also be decreasing in $\mathbb{E}\left[\sum \boldsymbol{X}_i\right]$. We can then try to model $\hat{g}_i$ as some decreasing function in $\sum \boldsymbol{x}_i \geq 0$, where the relation between $\rho_i\left(\boldsymbol{X}\right)$ and $\mathbb{E}\left[\sum \boldsymbol{X}_i\right]$ is shaped by η_i. A function that fulfills these specifications is

$$\hat{g}_i\left(\boldsymbol{x}, \eta_i\right) = \frac{1}{\left(1 + \left(k_i^{-1} \sum \boldsymbol{x}_i\right)^{\eta_i}\right)^{1/\eta_i}} \qquad \eta_i > 0, \ \forall i \in \mathcal{Q}, \tag{3.31}$$

which can be recognized as the *inverse p-norm* of the vector function $v\left(z\right) = \left[1, \ z\right]$ where $z = k_i \left(\sum \boldsymbol{x}_i\right)^{-1}$ and where $p = \eta$. This choice of function is motivated by the following desirable properties. First, the function is monotonic.

Proposition 3.4
$\hat{g}_i\left(\boldsymbol{x}, \eta_i\right)$ is monotonically decreasing in $\sum \boldsymbol{x}_i \geq 0$ if $\eta_i > 0$, and monotonically nondecreasing in $\eta_i \in \mathbb{R}_+^{|\mathcal{Q}|} \ \forall i \in \mathcal{Q}$.

Proof. Differentiating the inverse p-norm of $v(z)$ yields

$$\frac{d}{dz}\|v(z)\|_\eta^{-1} = \frac{d}{dz}\left(1 + z^\eta\right)^{-\frac{1}{\eta}} = -z^{\eta-1}\left(1 + z^\eta\right)^{-\frac{\eta+1}{\eta}}, \tag{3.32}$$

which is negative for all $z \geq 0$ and $\eta > 0$. Hence, $\hat{g}_i$ is monotonically decreasing in $\sum \boldsymbol{x}_i \geq 0$ for all $\eta_i > 0$ as $z = k_i \left(\sum \boldsymbol{x}_i\right)^{-1} > 0$ in this domain. Further, a classic result for p-norms gives that $\|\boldsymbol{x}\|_\eta \geq \|\boldsymbol{x}\|_{\eta+a}$ for any $\eta > 0, a \geq 0$. Inverting both sides yields that $\hat{g}_i$ is monotonically nondecreasing in η_i. □

Furthermore, notice that we in the limit $\eta_i \to +\infty$ regain the original g_i of the mean-field fluid model, hence the existence of an optimal $\boldsymbol{\eta}$ can be guaranteed.

Proposition 3.5
For any valid $\boldsymbol{X}$, $\exists \boldsymbol{\eta}^o$ for (3.31) such that equality holds in (3.27), $\forall i \in \mathcal{Q}$.

Proof. For any $\boldsymbol{x} \geq 0$, the upper and lower bounds of $\hat{g}_i$ in η become $\sup_p \hat{g}_i\left(\boldsymbol{x}, \eta\right) = g_i\left(\boldsymbol{x}\right)$ and $\inf_\eta \hat{g}_i\left(\boldsymbol{x}, \eta\right) = 0$ if $\sum \boldsymbol{x}_i > 0$. As $k_i\rho_i(\boldsymbol{X})/\mathbb{E}\left[\sum X_i\right]$ is bounded by these two quantities as seen in (3.30), and (3.31) is continuous in $\boldsymbol{\eta}$, such a $\boldsymbol{\eta}^o$ must exist. □

Finally, any choice of $\eta_i \geq \eta_i^o$ is guaranteed to yield a better approximation in (3.27) than the standard mean-field model.

Proposition 3.6
For any valid $\boldsymbol{X}$, if $\eta_i^o \leq \eta_i$ then

$$\left|\frac{k_i\rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} - \hat{g}_i\left(\mathbb{E}\left[\boldsymbol{X}\right], \eta_i\right)\right| \leq \left|\frac{k_i\rho_i(\boldsymbol{X})}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} - g_i\left(\mathbb{E}\left[\boldsymbol{X}\right]\right)\right|. \tag{3.33}$$

Proof. As $\hat{g}_i$ is monotonically nondecreasing in η_i and $\hat{g}_i = g_i$ if $\eta_i \to +\infty$, we get that if $\eta_i^o \leq \eta_i$ then $\hat{g}_i\left(\mathbb{E}\left[\boldsymbol{X}\right], \eta_i^o\right) \leq \hat{g}_i\left(\mathbb{E}\left[\boldsymbol{X}\right], \eta_i\right) \leq g_i\left(\mathbb{E}\left[\boldsymbol{X}\right]\right)$. □

Finally, (3.31) can be recognized as a smoothing of the two extreme cases (3.28) and (3.29). From this inverse p-norm smoothing of the processor share function, an improved mean-field fluid model can be created as

Definition 3.2—The Smoothed Mean-field Fluid Model
The smoothed mean-field fluid model is defined as the following ODE:

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= \boldsymbol{W}^T \hat{\theta}\left(\boldsymbol{x}, \boldsymbol{\eta}\right) + \boldsymbol{A}\boldsymbol{\lambda}, \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0), \quad \boldsymbol{\eta}(t) > \boldsymbol{0}, \end{aligned} \tag{3.34}$$

where $\hat{\theta}_a\left(\boldsymbol{x}, \boldsymbol{\eta}\right) = x_a \hat{g}_{Q(a)}\left(\boldsymbol{x}, \eta_{Q(a)}\right) \ \forall a \in \mathcal{S}$, and $\hat{g}_{Q(a)}\left(\boldsymbol{x}, \eta_{Q(a)}\right)$ is defined as (3.31).

Further, let $\boldsymbol{x}\left(t, \boldsymbol{\eta}\right)$ be the solution to (3.34) for a given $\boldsymbol{\eta}$, and $\boldsymbol{x}^*\left(\boldsymbol{\eta}\right)$ its fixed point. It is important to remember that a trajectory $\boldsymbol{\eta}^o(t)$ does not necessarily imply that $\mathbb{E}\left[\boldsymbol{X}\left(t\right)\right] = \boldsymbol{x}\left(t, \boldsymbol{\eta}^o\right)$, as $x_a \hat{g}_{Q(a)}\left(\boldsymbol{x}, \eta^o_{Q(a)}\right)$ only forms an approximation to (3.24) via a necessary condition.

Finding a Suitable Smoothing Value

Given a trajectory of $\boldsymbol{X}_i$ $i \in \mathcal{Q}$ in some time interval $t \in [0, T]$, the optimal value η_i^o will likely be different at different t. In general, we could expect that there is little hope of obtaining such a trajectory $\eta_i^o(t)$. A more suitable trade-off for usability would be to assign η_i some static value. From Proposition 3.6 we know that the approximation error to the necessary condition will be smaller if $\eta_i^o \leq \eta_i$, but we could probably expect *at least* a similar accuracy compared to the standard mean-field fluid model if we simply choose some fixed $\eta_i \approx \eta_i^o(t)\ \forall t \geq 0$ as long as $\nexists \eta_{\ll} \eta_i^o(t)\ \forall t \in [0, T]$.

Further, we can always quickly find a $\boldsymbol{\eta}^o$ such that the equality holds in (3.27) for stationary systems. To do this, the utilization must first be obtained which can be done via the utilization law (2.6) as $\rho_i = \sum_r \rho_{i,r} = \sum_r \lambda_{i,r} / \left(k_i \mu_{i,r}\right)$ if $\mu_{i,r} \forall r \in \mathcal{C}_i$ are known, otherwise it can be estimated by forming an empirical approximation of the queue length invariant probability distribution $\{\pi_c\}_{c \in 0:R}$ for some upper bound on R s.t. $\sum_{c=0}^{R} \pi_c \approx 1$. Each π_c can be obtained as the fraction of time spent with queue length c, i.e., $\pi_c \approx T^{-1} \int \mathbb{1}\left(\sum \boldsymbol{X}_i(t) = c\right) dt\ \forall c \in 0 : R$. From this we can approximate the utilization at queue i as $\rho_i \approx k_i^{-1} \sum_{c=0}^{R} \hat{\pi}_c \min(k_i, c)$.

Finding η_i for queue i then reduces to solving the optimization problem

$$\underset{\eta_i \in \mathbb{R}_+}{\text{minimize}} \quad f(\eta_i) = \left| \hat{g}_i\left(\mathbb{E}\left[\boldsymbol{X}\right], \eta_i\right) - \frac{k_i \hat{\rho}_i}{\mathbb{E}\left[\sum \boldsymbol{X}_i\right]} \right|, \tag{3.35}$$

which is guaranteed to have a solution $f(\eta_i^o) = 0$ via Proposition 3.5, and easily solved using e.g. bisection search as Proposition 3.4 gives us that $\hat{g}_i$ is monotonic in η_i. We denote $\boldsymbol{\eta}^o$ that is optimal for a stationary system $\boldsymbol{\eta}^*$.

Comments

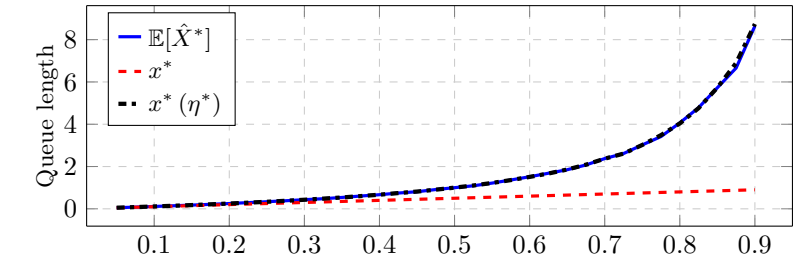
Compared to the other approaches for improving the mean-field approximation [Gast and Van Houdt, 2017; Randone et al., 2021], the smoothed mean-field fluid model (3.34) does not introduce any extra states. Instead, its computation time scales just as the standard mean-field fluid model with increasing state space. The two cited approaches are however more general in tackling arbitrary mean-field approximations, while our approach is tailored for multi-class PS queues. However, as the main ingredient of our approach is to approximate the spread of a stochastic process over a discontinuity with a smoothing function, the method should be extendable to any type of mean-field approximation that deals with discontinuities. This includes mean-field approximations over discriminatory PS or FCFS queues.

Furthermore, the smoothed model with $\boldsymbol{\eta}^*$ has strong similarities with the PSFFA method [Wei-Ping Wang et al., 1996] for nonstationary queues, see Section 2.1. In this method, the fluid model is represented as $\dot{x} = \mu G^{-1}(x) + \lambda$, where the function $G(\rho) = x^*$ is some approximation of the stationary queue length of the modelled queue. The smoothed model can thus be seen as a PSFFA model where $G^{-1}(\boldsymbol{x})$ is obtained using data-driven smoothing of the mean-field model.

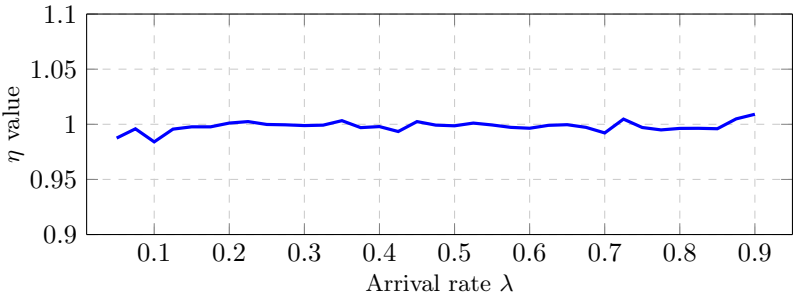
Testing Predictive Power on the Two Running Examples

To be usable as a performance model in cloud systems, it is paramount that the accuracy of $\hat{g}$ for an obtained $\boldsymbol{\eta}^*$ is robust to system perturbations, so that it can be used to accurately *predict* performance. Exactly what perturbation yields a robust $\hat{g}$ will be system dependent, but it should remain accurate as long as a perturbation does not radically alter $\boldsymbol{\eta}^*$. In Chapter 3.5, we will investigate the effect of three common perturbations on a simulated cloud application. For now, let us compare both stationary and transient mean queue lengths $\mathbb{E}\left[\hat{\boldsymbol{X}}\right]$ obtained from simulation with values from the smoothed mean-field fluid model for the two running examples introduced in the beginning of Chapter 3. Let the fixed point be defined as $\lim_{t\to+\infty} \boldsymbol{x}(t, \boldsymbol{\eta}) = \boldsymbol{x}^*(\boldsymbol{\eta})$ for the smoothed mean-field fluid model for some static value of $\boldsymbol{\eta}$.

Example 1:. Considering the M/M/1 queue, the results for the stationary mean queue length are shown in Figure 3.3 over an increasing sequence of $\lambda \in [0.05, 0.9]$. Here, it can be seen that the smoothed model manages to provide a good approximation of the stationary mean queue length. Further,



(a) Stationary mean queue length.



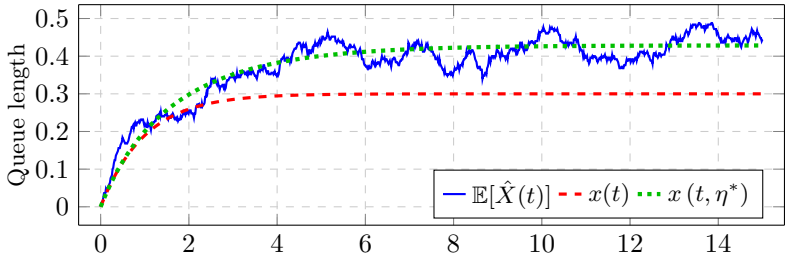
(b) Estimated η.

Figure 3.3 Comparison of the stationary mean queue length in Example 1 over different arrival rates, shown in (a) considering values obtained from simulation (blue line), the mean-field fluid model (red dashed) and the smoothed model using η^* estimated from data at every λ (black dash-dotted). In (b), η^* is displayed.

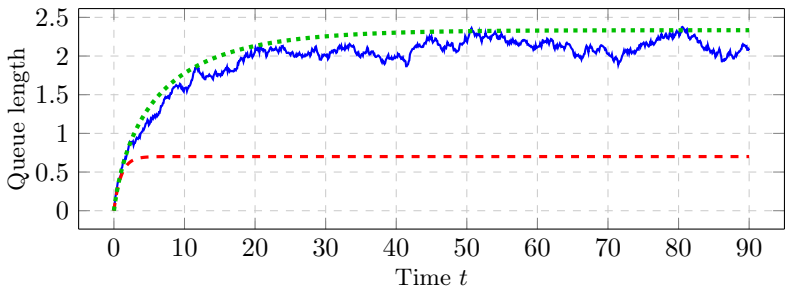
the η^* found is consistently near 1. A value of $p = 1$ yields $\dot{x} = -\frac{x}{x+1} + \lambda$, which corresponds to the well-known PSFFA model for M/M/1 queues, also known as the Tipper model [Tipper and Sundareshan, 1990]. The smoothed model thus manages to find the closed-form of this $G^{-1}(x)$, which is expected considering that $\hat{g}_a\left(\boldsymbol{x}, \eta_{Q(a)}\right)$ is monotone in $\eta_{Q(a)}$.

Furthermore, in Figure 3.4 the mean queue length transients are shown. As can be seen the smoothed mean-field fluid model manages to capture the transients well, in contrast to the nominal mean-field fluid model.

Example 2:. Considering the three cyclic queues, the results for the stationary mean queue lengths are shown in Figure 3.5 over an increasing sequence of $\mu_1 \in [0.05, 0.95]$. Here, both $\boldsymbol{\eta}^*$, which is estimated from data for every value of μ_1, and $\hat{\boldsymbol{\eta}}$ estimated at $\mu_1 = 0.2$ are considered. Fitting $\boldsymbol{\eta}$ when $\mu_1 = 0.2$ for the two PS queues yields that $\hat{\eta}_2 \approx 2.24$ and $\hat{\eta}_3 \approx 3.5$,



(a) Mean queue length transients for $\lambda = 0.3$.



(b) Mean queue length transients for $\lambda = 0.7$.

Figure 3.4 Comparison of the mean queue length transients in Example 1 for different values of λ. The blue lines shows the queue length estimated from 250 repeated simulations, the red dashed lines the mean-field fluid model, and the green dotted lines the smoothed mean-field fluid model with $\eta^* = 1$.

which results in the following smoothed version of the $\theta(\boldsymbol{x})$ function:

$$\hat{\theta}\left(\boldsymbol{x}, \hat{\boldsymbol{\eta}}\right) = \begin{bmatrix} x_1 \\ x_2 \cdot \left(1 + \left[4^{-1}\left(x_2 + x_3\right)\right]^{2.24}\right)^{-1/2.24} \\ x_3 \cdot \left(1 + \left[4^{-1}\left(x_2 + x_3\right)\right]^{2.24}\right)^{-1/2.24} \\ x_4 \cdot \left(1 + \left[8^{-1}\left(x_4 + x_5\right)\right]^{3.5}\right)^{-1/3.5} \\ x_5 \cdot \left(1 + \left[8^{-1}\left(x_4 + x_5\right)\right]^{3.5}\right)^{-1/3.5} \end{bmatrix}. \tag{3.36}$$

As can be seen, using $\boldsymbol{\eta}^*$ yields a very good approximation of the stationary mean queue lengths but creates a dependency on data generated for μ_1. Instead, using $\hat{\boldsymbol{\eta}}$ manages to provide a good prediction of the mean queue

lengths for each perturbation of μ_1 from the nominal value of $\mu_1 = 0.2$, despite the apparent dissimilarity between $\hat{\boldsymbol{\eta}}$ and $\boldsymbol{\eta}^*$. This hints at some form of insensitivity in how similar $\boldsymbol{\eta}$ and $\boldsymbol{\eta}^*$ need to be in order to provide an adequate prediction. Moreover, using $\hat{\boldsymbol{\eta}}$ outperforms the standard mean-field model by a large margin for most μ_1, which is not surprising given that the mean-field fluid model can be obtained by setting $\boldsymbol{\eta} = +\infty$.

Further, the mean queue lengths transients are shown in Figure 3.6. Similarly to Example 1, the smoothed mean-field fluid model manages to capture the transients well both at $\mu_1 = 0.2$ and for predictions over different μ_1. Here, the performance of the nominal mean-field fluid model is better than in Example 1. It manages to capture the initial quick transients, but then converges towards the wrong stationary value.

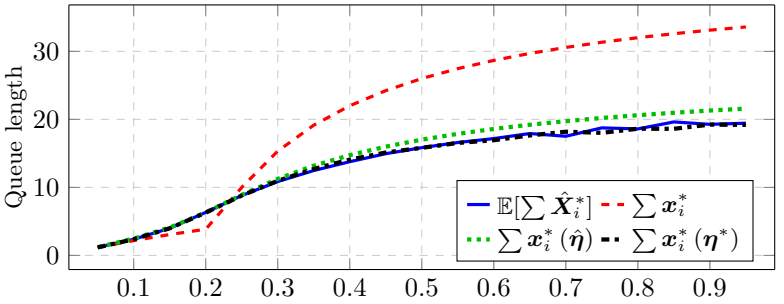
3.4 Closed-Form Response Time CDF Approximation

Various percentiles of the response time are important metrics when modeling the performance of systems in cloud computing. As shown in [Pérez and Casale, 2013; Pérez and Casale, 2017] and later refined in [Zhu et al., 2020], for a closed PS queueing network at stationarity, it is possible to obtain better approximations to the percentiles than simply using the Chebyshev bound by approximating the response time CDF as the solution of a state-space extended mean-field model. Shortly explained, the extended model can be created by duplicating the states of interest with the same out-connections as the original states but with no in-connections. By then calculating the trajectory of $\boldsymbol{x}$, it is possible to obtain the relative loss of mass at the states of interest at any $t \geq 0$, which can be used to approximate the CDF. We refer to the original papers for a more thorough explanation.

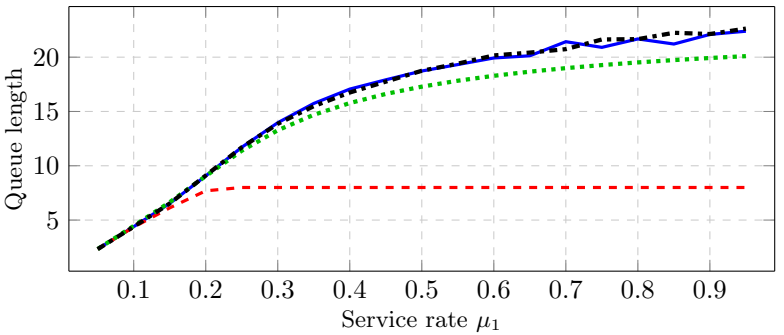
However, it is possible to obtain a closed-form approximation of the entire response time CDF of a mixed PS queueing network at stationarity over almost any subset of classes and queues. Start by considering the case of a request that at $t = 0$ enters class r in queue i. The request will enter any $a \in \mathcal{S}_{i,r}$ with the probability $\zeta_a^{i,r}$ and be served at a speed of $\frac{\min(k_i, \sum \boldsymbol{X}_i(t))}{\sum \boldsymbol{X}_i(t)} \Psi_{a,a}^{i,r}$, before departing (i, r) or entering another state in $\mathcal{S}_{i,r}$. Via the Chapman–Kolmogorov equation [Gardiner, 1985], the probability $\boldsymbol{\pi} \in \mathbb{R}_+^{|S_{i,r}| \times 1}$ of finding the request in a given state at time t then evolves according to

$$\dot{\boldsymbol{\pi}}(t) = \left(\Psi^{i,r}\right)^T \frac{\min\left(k_i, \sum \boldsymbol{X}_i(t)\right)}{\sum \boldsymbol{X}_i(t)} \boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \zeta^{i,r}. \tag{3.37}$$

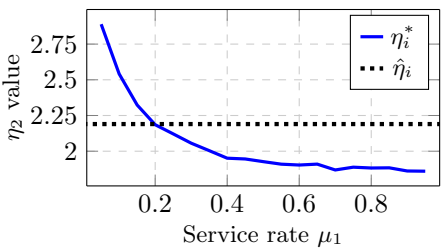
Due to its dependence on $\sum \boldsymbol{X}_i(t)$, $\boldsymbol{\pi}$ is itself a stochastic variable. Obtaining its mean becomes tricky as we have little information regarding the statistics of $\sum \boldsymbol{X}_i(t)$. Instead, we can create a manageable approximation by assuming



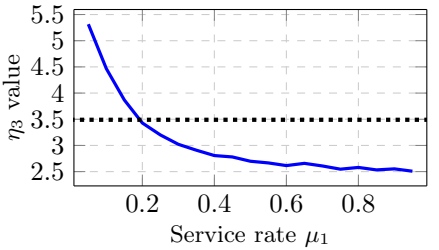
(a) Stationary mean queue length in PS queue $i = 2$.



(b) Stationary mean queue length in PS queue $i = 3$.

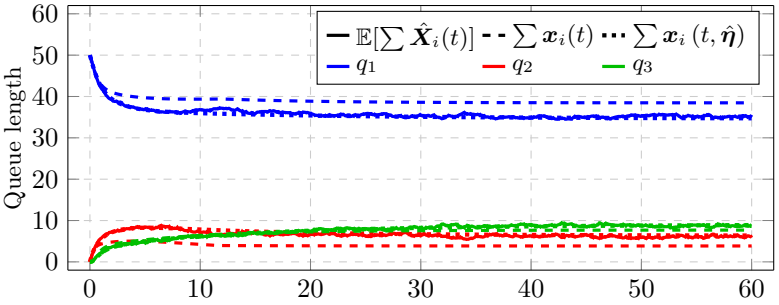


(c) Estimated η_i for queue $i = 2$.

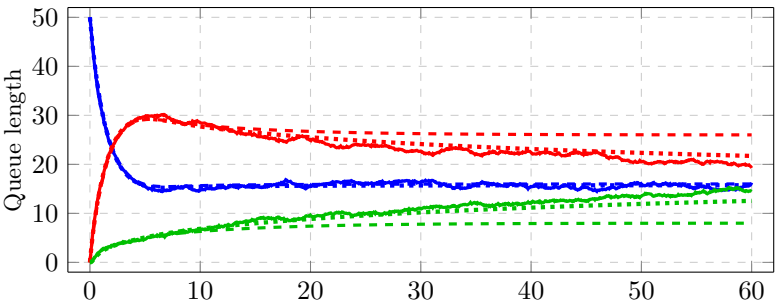


(d) Estimated η_i for queue $i = 3$.

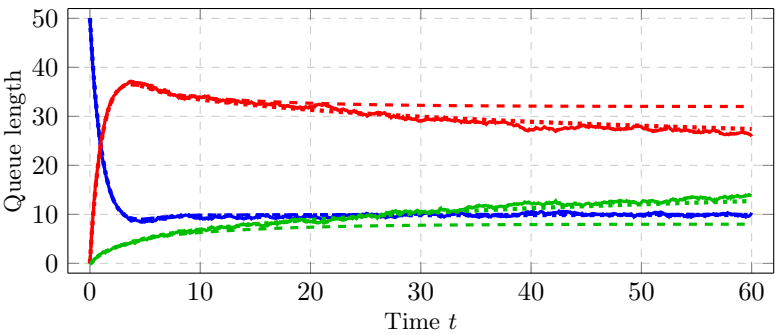
Figure 3.5 Comparison of the stationary mean queue lengths in Example 2 over different service rates for queues 2 and 3, shown in (a) and (b) considering values obtained from simulation (blue line), the mean-field fluid model (red dashed) and the smoothed model using both $\hat{\boldsymbol{\eta}}$ estimated from data at $\mu_1 = 0.2$ (green dotted) and $\boldsymbol{\eta}^*$ estimated from data at every μ_1 (black dash-dotted). In (c) and (d), $\boldsymbol{\eta}^*$ (blue line) and $\hat{\boldsymbol{\eta}}$ (black dotted) are displayed. As $\boldsymbol{X}_i$ where $i \in \mathcal{Q}$ gives the vector of requests in each state belonging to i, the total queue length is given as $\sum \boldsymbol{X}_i$.



(a) Mean queue length transients for $\mu_1 = 0.2$.



(b) Mean queue length transients for $\mu_1 = 0.5$.



(c) Mean queue length transients for $\mu_1 = 0.8$.

Figure 3.6 Comparison of the mean queue length transients in Example 2 for different values of μ_1. At $t = 0$, there are 50 requests in queue 1. The full lines shows the queue lengths estimated from 100 repeated simulations, the dashed lines the mean-field fluid model and the dotted lines the smoothed mean-field fluid model with $\hat{\boldsymbol{\eta}}$ estimated from data at $\mu_1 = 0.2$. The blue lines shows the transient values for queue 1, the red lines for queue 2 and the green lines for queue 3.

stationarity and that $\sum \boldsymbol{X}_i^*(t)$ and $\boldsymbol{\pi}(t)$ are independent, which yields that

$$\mathbb{E}\left[\dot{\boldsymbol{\pi}}(t)\right] \approx \left(\Psi^{i,r}\right)^T \sum_{c\geq 0} \mathbb{P}\left(\sum \boldsymbol{X}_i^* = c\right) \frac{\min\left(k_i, c\right)}{c} \mathbb{E}\left[\boldsymbol{\pi}(t)\right]. \tag{3.38}$$

The sum $\sum_{c\geq 0} \mathbb{P}\left(\sum \boldsymbol{X}_i^* = c\right) \frac{\min(k_i,c)}{c}$ can be recognized as $\mathbb{E}\left[g_i(\boldsymbol{X}^*)\right]$, thus the approximation can be motivated as assuming that the request receives the mean processor share during its duration. Using the smoothed mean-field fluid model from Chapter 3.3, we readily have an approximation of it via (3.31). This yields that

$$\mathbb{E}\left[\dot{\boldsymbol{\pi}}(t)\right] \approx \left(\Psi^{i,r}\right)^T \hat{g}_i\left[\boldsymbol{x}^*\left(\boldsymbol{\eta}\right), \boldsymbol{\eta}\right] \mathbb{E}\left[\boldsymbol{\pi}(t)\right]. \tag{3.39}$$

For simplicity, let $\hat{g}_i\left(\boldsymbol{\eta}\right) = \hat{g}_i\left[\boldsymbol{x}^*\left(\boldsymbol{\eta}\right), \boldsymbol{\eta}\right]$. The mean derivative can thus be approximated by a system of first-order linear ODEs, which yields the following closed-form solution:

$$\mathbb{E}\left[\boldsymbol{\pi}(t)\right] \approx \exp\left[\left(\Psi^{i,r}\right)^T \hat{g}_i\left(\boldsymbol{\eta}\right) t\right] \boldsymbol{\pi}(0), \tag{3.40}$$

where $\exp[\cdot]$ is the matrix exponential. Let $\tau \in \mathbb{R}_+$ be a stochastic variable representing the response times at stationarity over (i,r). Its CDF can be expressed as $\Phi_{i,r}(t) = \mathbb{P}\left(\tau \leq t\right) = 1 - \mathbb{P}\left(\tau > t\right)$. The probability of τ being larger than some t is the same as the probability of the request remaining in any $a \in \mathcal{S}_{i,r}$ at that time. Thus, the response time CDF can be approximated as

$$\Phi_{i,r}\left(t \mid \boldsymbol{\eta}\right) \approx 1 - \boldsymbol{\pi}(0)^T \exp\left[\hat{g}_i\left(\boldsymbol{\eta}\right) \Psi^{i,r} t\right] \mathbb{1}. \tag{3.41}$$

Using the same technique, it is further possible to obtain a closed-form approximation of the response time CDF over an almost arbitrary subset of classes and queues. Let $\mathcal{C}_R \subset \mathcal{C}$ denote such a subset; it is valid as long as for each closed chain $\gamma_c \in \Gamma$, $\exists (i,r) \in \gamma_c \; s.t. \; r \notin \mathcal{C}_R$. This is needed, as otherwise situations might occur where a request would never depart $\mathcal{C}_R$. Let $\tau_{\mathcal{C}_R} \in \mathbb{R}_+$ be a stochastic variable that describes the response time in stationarity over $\mathcal{C}_R$. Define $\beta \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ as the probability of entering specific states in $\mathcal{C}_R$ at $t = 0$, i.e., $\sum \beta = 1$, $\beta_r \geq 0$ if $r \in \mathcal{C}_R$ else $\beta_r = 0$. Now reintroduce $\boldsymbol{\pi}(t) \in \mathbb{R}_+^{|\mathcal{S}|\times 1}$ to describe the evolution of the probability that a request remains in $\mathcal{S}$ over time.

To only consider the probability over states in $\mathcal{C}_R$, we introduce the matrix $\boldsymbol{P}_R \in \mathbb{R}_+^{|\mathcal{C}|\times|\mathcal{C}|}$ that is given the same values for all routing probabilities in $\boldsymbol{P}$ that occur between classes in $\mathcal{C}_R$, i.e., $\forall r, s \in \mathcal{C} \;\; \left(\boldsymbol{P}_R\right)_{r,s} = \boldsymbol{P}_{r,s}$ if $r, s \in \mathcal{C}_R$ otherwise $\left(\boldsymbol{P}_R\right)_{r,s} = 0$. Furthermore, let $\boldsymbol{W}_R = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}_R\boldsymbol{A}^T$. In order to

only consider arrivals to $\mathcal{C}_R$, let $\boldsymbol{\pi}(0) = \boldsymbol{A}\beta$. The probability then evolves forward in time as

$$\dot{\boldsymbol{\pi}}(t) = \boldsymbol{W}_R^T D^{g(\boldsymbol{X}(t))} \boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \boldsymbol{A}\beta, \tag{3.42}$$

where $g(\boldsymbol{X}(t))$ is the processor share at some time t, and $D \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}$ is a diagonal matrix s.t. $D_{a,a}^{g(\boldsymbol{X}(t))} = g_{Q(a)}(\boldsymbol{X}(t))\ \forall a \in \mathcal{S}$. Following the same logic as for the single class/queue case, the expected value of the time derivative of $\boldsymbol{\pi}$ can be approximated as

$$\mathbb{E}\left[\dot{\boldsymbol{\pi}}(t)\right] \approx \boldsymbol{W}_R^T D^{\hat{g}(\boldsymbol{\eta})} \mathbb{E}\left[\boldsymbol{\pi}(t)\right], \tag{3.43}$$

which by fixing β, yields the following CDF approximation of $\tau_{\mathcal{C}_R}$:

$$\Phi_{\mathcal{C}_R}(t \mid \beta) \approx 1 - \beta^T \boldsymbol{A}^T \exp\left[D^{\hat{g}(\boldsymbol{\eta})} \boldsymbol{W}_R t\right] \mathbb{1}. \tag{3.44}$$

Thus far we have only considered the case of a single request where we are free to set β, but in order to approximate the CDF of $\tau_{\mathcal{C}_R}$ for an arbitrary request in some system the corresponding β needs to be obtained. First, let $\mathcal{C}_R^c$ denote the complement of $\mathcal{C}_R$ in $\mathcal{C}$. Introduce $\boldsymbol{\mu}_d \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ as the average outflow at stationarity from each class in $\mathcal{C}$. It is given as

$$\boldsymbol{\mu}_d = \boldsymbol{B}^T \mathbb{E}\left[D^{g(\boldsymbol{X}^*)} \boldsymbol{X}^*\right], \tag{3.45}$$

which can be approximated as

$$\hat{\boldsymbol{\mu}}_d(\boldsymbol{\eta}) = \boldsymbol{B}^T D^{\hat{g}(\boldsymbol{\eta})} \boldsymbol{x}^*(\boldsymbol{\eta}). \tag{3.46}$$

Let $\boldsymbol{P}_{\lambda(\mathcal{C}_R^c)} \in \mathbb{R}_+^{|\mathcal{C}_R^c|\times|\mathcal{C}_R|}$ be a submatrix of $\boldsymbol{P}$ describing the transitions from $\mathcal{C}_R^c$ to $\mathcal{C}_R$. The total mean external flow to $\mathcal{C}_R$ can then be approximated as

$$\hat{\beta}_{\mathcal{C}_R}^v(\boldsymbol{\eta}) = \boldsymbol{P}_{\lambda(\mathcal{C}_R^c)}^T \left[\hat{\boldsymbol{\mu}}_d(\boldsymbol{\eta})\right]_{\mathcal{C}_R^c} + \boldsymbol{\lambda}_{\mathcal{C}_R}. \tag{3.47}$$

Thus $\forall r \in \mathcal{C}$, $\hat{\beta}_r = \left(\hat{\beta}_{\mathcal{C}_R}^v\right)_r / \sum \hat{\beta}_{\mathcal{C}_R}^v$ if $r \in \mathcal{C}_R$ else $\hat{\beta}_r = 0$ if $r \in \mathcal{C}_R^c$, and a closed-form approximation of $\Phi_{\mathcal{C}_R}(t \mid \beta)$ for an arbitrary request entering $\mathcal{C}_R$ is given by

$$\Theta_{\mathcal{C}_R}(t \mid \boldsymbol{\eta}) = 1 - \hat{\beta}(\boldsymbol{\eta})^T \boldsymbol{A}^T \exp\left[D^{\hat{g}(\boldsymbol{\eta})} \boldsymbol{W}_R t\right] \mathbb{1}. \tag{3.48}$$

The validity of this approximation is determined in the following proposition.

Proposition 3.7
The introduced $\Theta_{\mathcal{C}_R}(t \mid \boldsymbol{\eta})$ is a valid CDF for all valid choices of $\mathcal{C}_R$ and flow matrices as defined in Theorem 3.1.

Proof. For a function to be a valid CDF, it must satisfy the following three criteria; (i) $\Phi(t)$ should be monotonically nondecreasing in t, (ii) $\Phi(0) = 0$, and (iii) $\lim_{t\to+\infty} \Phi(t) = 1$.

(ii) can be seen to hold right away as $\Theta_{\mathcal{C}_R}(0 \mid \boldsymbol{\eta}) = 1 - \hat{\beta}(\boldsymbol{\eta})^T \mathbb{1} = 0$ by construction. To show the remaining two conditions, we first introduce $\boldsymbol{\pi}_R \in \mathbb{R}_+^{|\mathcal{C}_R|\times 1}$ as the approximated probability vector for the classes in $\mathcal{C}_R$, and $\Lambda \in \mathbb{R}^{|\mathcal{C}_R|\times|\mathcal{C}_R|}$ the submatrix of $D^{\hat{g}(\boldsymbol{\eta})}\boldsymbol{W}_R$ that describes the change of $\boldsymbol{\pi}_R$ such that $\dot{\boldsymbol{\pi}}_R = \Lambda^T \boldsymbol{\pi}_R$ and by construction (3.48) $= 1 - \boldsymbol{\pi}_R(t)^T \mathbb{1}$.

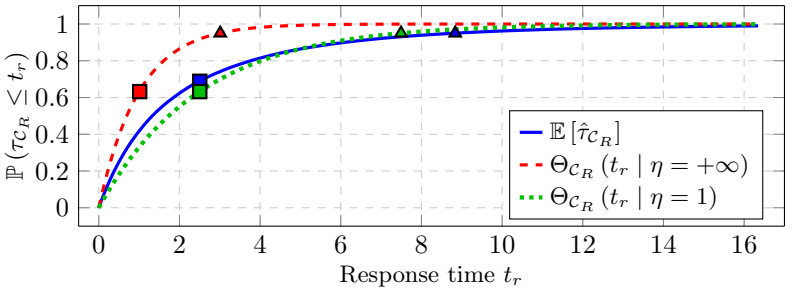
Now, let Γ_R be the set of chains present in $\mathcal{C}_R$. Each chain $\gamma_R \in \Gamma_R$ is a subset of a chain $\gamma \in \Gamma$. As $\gamma_c \in \Gamma$ are considered irreducible, the corresponding $\gamma_R \in \Gamma_R$ will be open and transient. Thus, all $\gamma \in \Gamma_R$ are open and transient, and since $D^{\hat{g}(\boldsymbol{\eta})}$ only imposes a positive scaling constant for each row in $\boldsymbol{W}_R$, Proposition 3.3 applies to all matrices in the partition $\{\Lambda_{\gamma_R}\}_{\gamma_R\in\Gamma_R}$ and subsequently will have eigenvalues with strictly negative real values. This implies that Λ only has eigenvalues with strictly negative real values, and thus the time limit of $\boldsymbol{\pi}_R$ is given by $\lim_{t\to+\infty} \boldsymbol{\pi}_R(t) = \mathbf{0}$, which gives that $\lim_{t\to+\infty} \Theta(t \mid \cdot) = 1 - 0$, proving (iii). Furthermore, as the row sums of Λ are nonpositive, $\boldsymbol{\pi}_R(t)^T \mathbb{1}$ is moreover monotonically nonincreasing, thus proving (i). This can be seen by considering that for any valid $\boldsymbol{\pi}$ we get that $\frac{d}{dt}\boldsymbol{\pi}_R^T \mathbb{1} = \sum_j \left(\sum_i \Lambda_{j,i}\right)(\boldsymbol{\pi}_R)_j \leq 0$. □

In practice, akin to the proof above, the CDF approximation can be more efficiently formed by first removing all elements in the involved matrices and vectors corresponding to state whose classes are not in $\mathcal{C}_R$. By construction, these states will not receive any probability mass and no probability mass will start here, hence they can simply be removed.

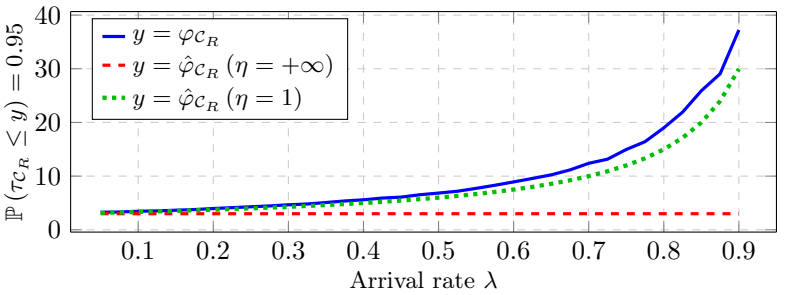
The accuracy of the response time CDF approximation mainly depends on how accurate the required assumption is, which states that every request receives the mean processor share. Thus in general, the larger the variability in $\boldsymbol{X}(t)$ becomes, the worse the approximation will be, as we will get a larger portion of requests whose received processor share differs significantly from the mean. However, this should imply that the approximation becomes more accurate the larger the system size becomes.

CDF Approximation Over the Two Running Examples

We will now compare both the CDF and 95th percentile of $\tau_{\mathcal{C}_R}$ from simulated data with the closed-form approximation on our two running examples. Denote $\varphi_{\mathcal{C}_R}$ as the 95th percentile of $\tau_{\mathcal{C}_R}$, and $\hat{\varphi}_{\mathcal{C}_R}(\boldsymbol{\eta})$ its approximation via (3.48). We will consider $\mathbb{E}[g(\boldsymbol{X}^*)]$, approximated both with $\hat{g}(\boldsymbol{\eta})$ from the smoothed model, and with $g(\boldsymbol{x}^*)$ from the standard mean-field fluid model which should yield similar results as the simulation method used in [Pérez and Casale, 2013; Pérez and Casale, 2017; Zhu et al., 2020].



(a) Response time CDF.



(b) 95th percentile of response times.

Figure 3.7 Comparison of (a) the CDF of $\tau_{\mathcal{C}_R}$ for $\lambda = 0.6$, and (b) $\varphi_{\mathcal{C}_R}$ over increasing λ, for Example 1 considering values obtained from simulation (blue line), the mean-field fluid model (red dashed) and the smoothed model with $\eta = 1$ (green dotted). The squares and triangles display the mean value and 95th percentile of the corresponding CDF.

Example 1:. Considering the M/M/1 system, the response time CDF can be obtained directly from (3.41), which for the given $\eta^* \approx 1$ yields that

$$\Phi(t \mid \eta = 1) \approx 1 - \exp\left[-\frac{t}{1 + x^*}\right]. \tag{3.49}$$

This holds for any fixed point of the smoothed model x^* generated by an arbitrary $\lambda < 1$. The resulting CDF approximations are displayed in Figure 3.7. As can be seen, using the standard mean-field fluid model to approximate $\mathbb{E}[g(\boldsymbol{X}^*)]$ results in bad accuracy, whereas the smoothed model with the well-chosen $\eta = 1$ gives an accurate approximation. The accuracy decreases the larger λ becomes, due to the increasing variability in $\boldsymbol{X}^*$.

Example 2:. For the closed cyclic network, we chose $\mathcal{C}_R$ to include the classes in q_2 and q_3, hence the response time CDF will be approximated over the two PS queues q_2, q_3, i.e., $\mathcal{C}_R = \mathcal{C}_{q_2} \cup \mathcal{C}_{q_3}$ and $\mathcal{C}_R^c = \mathcal{C}_{q_1}$. See Appendix B on how to obtain the CDF approximation for this example.

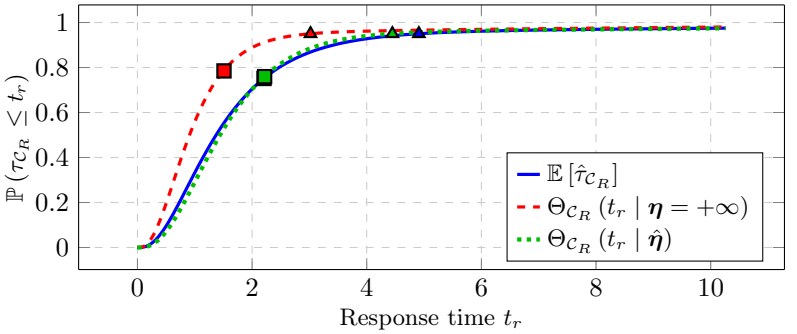
The resulting CDF approximations compared to simulated data are displayed in Figure 3.8. As can be seen, the smoothed mean-field fluid model manages to provide both an accurate CDF approximation at $\mu_1 = 0.2$ where $\boldsymbol{\eta}$ was estimated and a prediction of $\varphi_{\mathcal{C}_R}$ when perturbing μ_1. In contrary to Example 1, using the standard mean-field fluid model to approximate β and $\mathbb{E}\left[g\left(\boldsymbol{X}^*\right)\right]$ seems to produce a fairly accurate approximation. However, this should be taken with a grain of salt, as considering each queue separately instead leads to large errors as can be seen in Figure 3.8(c) and 3.8(d).

3.5 Evaluation on a Simulated Cloud Application

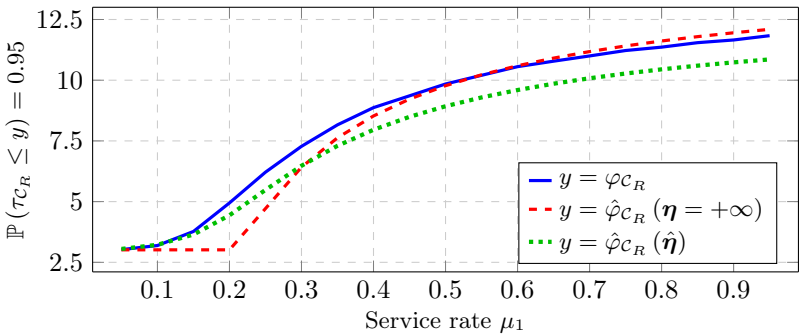
As demonstrated by the running examples, the proposed smoothed mean-field fluid model gives an increased accuracy both when approximating and predicting (under changing load) the stationary and transient mean queue lengths and CDFs of the response times. In general, we have noticed that it is easy to generate other example systems where the smoothed model outperforms the standard in terms of approximation error and systems where the models provide similar accuracy. As long as for some $i \in \mathcal{Q}$, the PMF $\boldsymbol{X}_i^*$ has large support on both sides of k_i, the smoothed model using $\boldsymbol{\eta}^*$ seems to provide a clear benefit. However, the question remains how these results hold up for prediction when the system is perturbed. To examine this, we studied a more advanced queueing network, representing the two-tier cloud application shown in Figure 3.9. To limit the set of potential types of perturbations, we considered three relevant types that are common in cloud computing, namely *workload change* (changing the rate of user arrivals), *horizontal scaling* (changing the number of replicas to an existing service), and *vertical scaling* (changing the allocated resources to a replica of an existing service) [Vaquero et al., 2011].

The Simulation Experiment

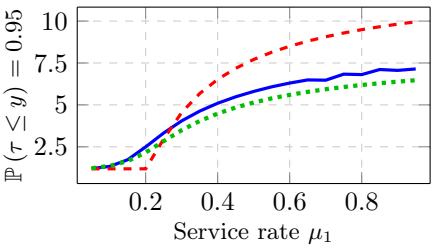
The two-tier system shown in Figure 3.9 comprises two services, denoted `frontend` f and `backend` b. The `frontend` service has two replicas, while the `backend` has four. We assumed a random load balancing strategy such that each request making a user-to-service or service-to-service transition will have an equal probability to end up in any replica of the target service. Using this two-tier system is a population of N clients c as well as occasional external users e. Any request from either a client or an external user is processed first at f, then at b, and then finally at f again. When a client's request



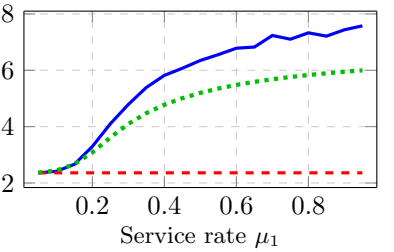
(a) Response time CDF considering both q_2 and q_3.



(b) 95th percentile of response times considering both q_2 and q_3.



(c) 95th percentiles of RT in q_2.



(d) 95th percentiles of RT in q_3.

Figure 3.8 Comparison of (a) the CDF of $\tau_{\mathcal{C}_R}$ for $\mu_1 = 0.2$, and (b) $\varphi_{\mathcal{C}_R}$ over increasing μ_1, for $\mathcal{C}_R = \mathcal{C}_2 \cup \mathcal{C}_3$ in Example 2 considering values obtained from simulation (blue line), the mean-field fluid model (red dashed) and the smoothed model with $\hat{\eta}$ estimated from data at $\mu_1 = 0.2$ (green dotted). The squares and triangles display the mean value and 95th percentile of the corresponding CDF. In (c) and (d), $\varphi_{\mathcal{C}_2}$ and $\varphi_{\mathcal{C}_3}$ are shown respectively over increasing μ_1. Here, the same legend as in (b) applies, but where $\mathcal{C}_R$ is either $\mathcal{C}_2$ or $\mathcal{C}_3$.

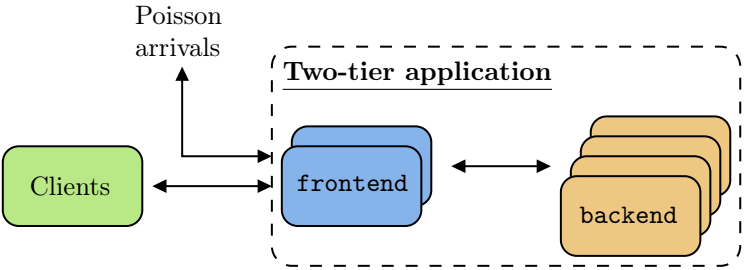


Figure 3.9 Example two-tier cloud application.

completes a trip through the two-tier system, the client will wait a random time before sending a new request. Requests from external users are assumed to be Poisson arrivals, and when completed, they depart from the network entirely.

By letting the waiting clients be represented by a single-class INF queue and f and b by PS queues with four and two classes, respectively, the two-tier system can be translated into a complex mixed PS queueing network. The network will contain one open and one closed chain corresponding to requests from e and c, who will make the following transitions:

$$\text{open chain:} \quad e \to f_1^i \to b_1^j \to f_2^k \to e \qquad i,k \in [1,4],\ j \in [1,2],$$
$$\text{closed chain:} \quad c \to f_3^i \to b_2^j \to f_4^k \to c \qquad i,k \in [1,4],\ j \in [1,2].$$

The service times of each class in each service are characterized by a mean $1/\mu$ and a coefficient of variation $\sigma\mu$ (CoV), which are used to fit the corresponding phase-type distributions. The baseline $1/\mu$ and $\sigma\mu$ that were used are shown in Table 3.1; these particular values were chosen to provide an interesting system, i.e., classes of both light- and heavy-tailed distributions and a similar mean in both tiers. For each replica, $1/\mu$ and $\sigma\mu$ were further scaled elementwise with random variables uniformly sampled from $[0.8, 1.2]$ to model heterogeneity. In order to provide an environment where the performance of the two fluid models differs, we chose the client population $N = 50$, mean interarrival time of the external arrivals $1/\lambda = 10$, and mean time and CoV for the clients as $1/\mu_c = 25$, $\sigma_c\mu_c = 1$.

Table 3.1 Baseline values for the two services.

	`frontend`	`backend`
k	8	4
$1/\mu$	[8.0, 8.0, 8.0, 8.0]	[16.0, 16.0]
$\sigma\mu$	[5.0, 0.5, 5.0, 0.5]	[10.0, 10.0]

For this system, data was generated via simulation from which a smoothing parameter $\boldsymbol{\eta}$ is estimated. To test how well $\hat{\boldsymbol{\eta}}$ generalizes in predicting the performance, a randomized simulation study was then performed in which random system perturbations were sampled from the following three types:

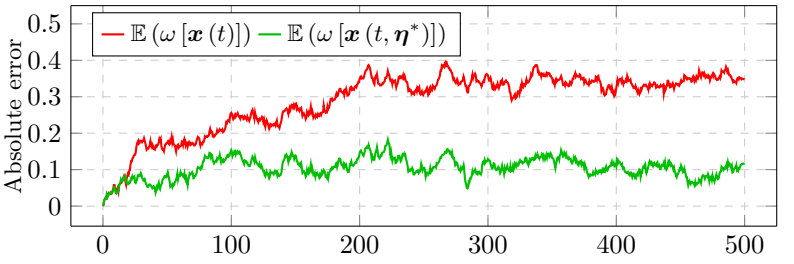
(i) **Workload change:** A new mean interarrival time between external users is drawn from $1/\lambda \sim U(2, 20)$, while a new client waiting time mean and CoV are drawn as $1/\mu_c \sim U(5, 50)$ and $\sigma_c \mu_c \sim C(\{U(0.1, 1.0), U(1.0, 10)\})$.

(ii) **Horizontal up-scaling:** With equal probability, a new f or b replica is created from the baselines with mean $\frac{1}{\mu} \odot u^{(1)}$ and CoV $\sigma\mu \odot u^{(2)}$ where $u_i^{(1,2)} \sim U(0.5, 1.5)$. This is done to emulate potential slowdown or speedup at the new replica location compared to the baseline. The replica is further given a smoothing value equal to the mean $\hat{\boldsymbol{\eta}}$ of all other replicas of the same service.

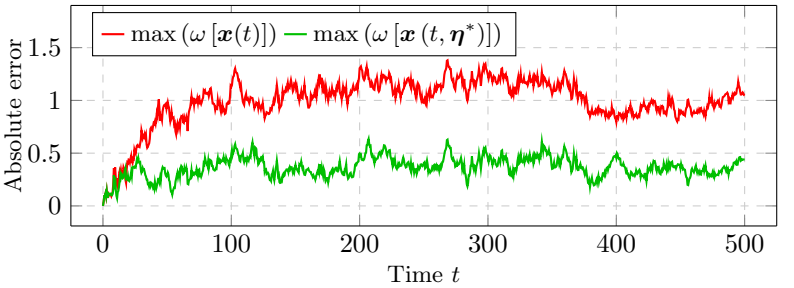
(iii) **Vertical scaling:** A random replica r is chosen among all `frontend` and `backend` replicas, and its mean service times scaled as $\frac{1}{\mu_r} \odot u$ where $u_i \sim C(\{U(0.2, 1.0), U(1.0, 5.0))\})$.

Here, $U(a, b)$ denotes the uniform distribution over the continuous interval $[a, b]$, and $C(\mathcal{A})$ the categorical distribution over each element in the set $\mathcal{A}$ with equal weights. The particular values were chosen such that the system remains stable for all possible perturbations. As there are only six possible combinations of horizontal downscaling, they were not part of the randomized simulation study but studied individually. In total, we generated 500 samples for each perturbation type, and each corresponding perturbed system was simulated for 500,000 time units to give enough datapoints to enable adequate approximations of stationary performance metrics for all possible perturbations.

Transient class populations. Before running the randomized simulation experiment, we examined the transient values of the request population in each class of the original system. Comparisons were made using the absolute error, since early t will have a disproportionately large relative error due to the noisy near-zero class population estimates. Furthermore, only classes in the application $\overline{\mathcal{C}}$ were considered as the client queue will otherwise have a large impact on this metric. We define the absolute error over the different class populations as $\omega(\boldsymbol{x}) = \{|\mathbb{E}[\sum \boldsymbol{X}_r(t)] - \boldsymbol{x}|\}_{r \in \overline{cC}}$. In Figure 3.10 the mean and maximum of these absolute errors are shown over time $t \in [0, 500]$. As can be seen, both the nominal and smoothed mean-field fluid models yield rather small errors when comparing them to the mean class population. However, the smoothed fluid model outperforms the nominal model at almost all times, as expected.



(a) Mean absolute error of the mean class population transients.



(b) Maximum absolute error of the mean class population transients.

Figure 3.10 The absolute errors between mean class population transients estimated from 250 repeated simulations and values obtained via the fluid model. The red lines shows the nominal fluid model, and the green lines the smoothed mean-field fluid model with $\boldsymbol{\eta}$ fitted at this operating condition. In (a), the mean absolute error over all classes in the simulated application is shown. In (b), the maximum absolute error is instead shown. For comparison, the mean class population at $t = 500$ is $\mathbb{E}[\sum \boldsymbol{X}_r] = 2.24$.

Implementation. The code for running the randomized simulation experiment is available on GitHub[1]. Included is also the code for the presented experiments with the running examples. It is mainly implemented in Julia[2], but uses the tool LINE [Casale, 2020] to construct the corresponding queueing network model, which in turn interfaces with JMT [Bertoli et al., 2009] to perform the simulations. The fluid models are solved using the `DifferentialEquations.jl` [Rackauckas and Nie, 2017] library together with the `LSODA` solver [Hindmarsh and Petzold, 2005].

[1] https://github.com/JohanRuuskanen/Performance2021_code/tree/thesis
[2] https://julialang.org/

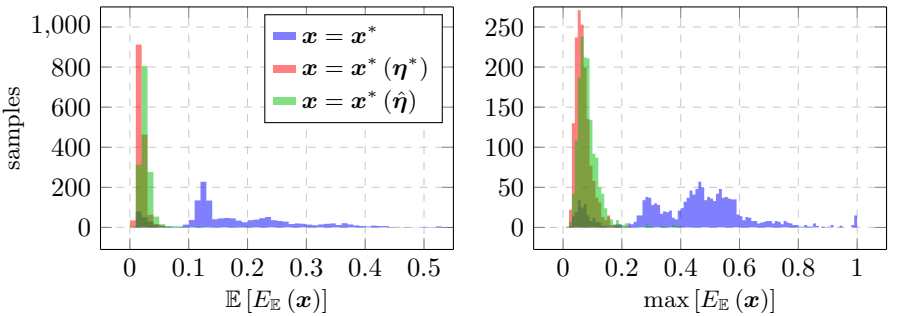
Results

To compare the accuracy, we considered the *relative error* (RE) of the stationary mean class population and the 95th percentile of the response times for every class in the network. Transient values were not considered as they would yield a very long computation time for every simulation experiment. We further considered $\varphi_{\mathcal{C}_R}$ over $\mathcal{C}_R = \bigcup_{i \in f^1_{3,4}, b^{1:4}_2} \mathcal{C}_i$, i.e., all classes in the first `frontend` replica and the four `backend` replicas in the original closed chain. Let $\boldsymbol{X}_r$ be the vector of phase states belonging to $r \in \mathcal{C}$. We define RE of stationary mean class population over class r as $\varepsilon_{\mathbb{E}}(\boldsymbol{x}, r) = |\mathbb{E}[\sum \boldsymbol{X}^*_r] - \sum \boldsymbol{x}_r| / \mathbb{E}[\sum \boldsymbol{X}^*_r]$, and RE of φ_r as $\varepsilon_{\varphi}(\boldsymbol{x}, r) = |\varphi_r - \hat{\varphi}_r(\boldsymbol{x})| / \varphi_r$. The set of REs over each $r \in \mathcal{C}$ for the stationary mean class population then becomes $E_{\mathbb{E}}(\boldsymbol{x}) = \{\varepsilon_{\mathbb{E}}(\boldsymbol{x}, r)\}_{r \in \mathcal{C}}$, and for the 95th percentile $E_{\varphi}(\boldsymbol{x}) = \{\varepsilon_{\varphi}(\boldsymbol{x}, r)\}_{r \in \mathcal{C}}$. For the baseline system without any perturbations, the obtained results are presented in Table 3.2. Unsurprisingly, the smoothed mean-field fluid model (indicated with $\boldsymbol{x} = \boldsymbol{x}^*(\hat{\eta})$) outperforms the standard model (indicated with $\boldsymbol{x} = \boldsymbol{x}^*$), obtaining both a lower mean RE and maximum RE over all classes in all queues.

The results of the randomized simulation experiment are presented as histograms in Figure 3.11. Each data point corresponds to a specific RE for a random perturbation, using either the standard mean-field model (blue), the smoothed model with $\boldsymbol{\eta}^*$ refitted for each perturbation (red), or the smoothed model with $\hat{\boldsymbol{\eta}}$ fitted to the baseline system (green). From it we can deduce mainly three things: firstly, the smoothed model in general outperforms the standard model by quite some margin; secondly, the response time percentiles are fairly accurate despite the approximation; and finally, the results using $\hat{\boldsymbol{\eta}}$ fitted to the baseline system seem robust to the three types of perturbations. It can also be seen that the standard mean-field fluid model sometimes yields good results, which most likely occurs when a random perturbation places the support of $\boldsymbol{X}^*_i$ on a single side of $k_i \; \forall i \in \mathcal{Q}$. The results of the horizontal down-scaling are left out, as they follow the same pattern. As the prediction accuracy of the smoothed model with $\hat{\boldsymbol{\eta}}$ is this robust for such a fairly complex queueing network, we generally expect a comparable result considering other queueing networks over similar perturbations.

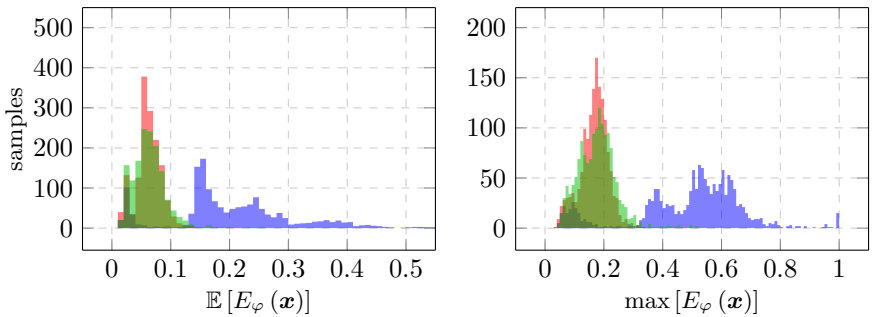
Table 3.2 Different RE metrics of the original system for the standard ($\boldsymbol{x}^*$) and smoothed ($\boldsymbol{x}^*(\boldsymbol{\eta}^*)$) models.

	$\mathbb{E}[E_{\mathbb{E}}(\boldsymbol{x})]$	$\max[E_{\mathbb{E}}(\boldsymbol{x})]$	$\mathbb{E}[E_{\varphi}(\boldsymbol{x})]$	$\max[E_{\varphi}(\boldsymbol{x})]$	$\varepsilon_{\varphi}(\boldsymbol{x} \mid \mathcal{C}_R)$
$\boldsymbol{x}^*$	0.23	0.54	0.25	0.61	0.32
$\boldsymbol{x}^*(\boldsymbol{\eta}^*)$	0.02	0.05	0.07	0.20	0.07



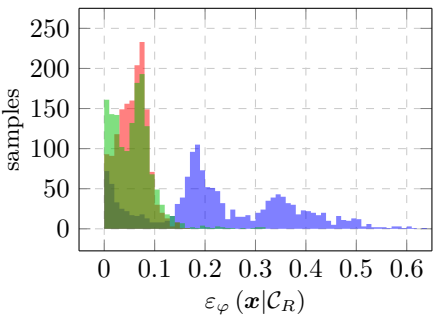
(a) Mean RE of avg. class pop.

(b) Max RE of avg. class pop.



(c) Mean RE of φ_r over every $r \in \mathcal{C}$.

(d) Max RE of φ_r over every $r \in \mathcal{C}$.



(e) RE of φ_r over $\mathcal{C}_R$.

Figure 3.11 Histograms with a bin width of 0.01 for five different RE metrics of 1500 random system perturbations, considering values obtained from the standard mean-field model (blue), the smoothed model with $\hat{\boldsymbol{\eta}}^*$ refitted for each perturbation (red), and the smoothed model with $\hat{\boldsymbol{\eta}}$ fitted to the baseline system (green). The bins are given a low opacity to better perceive overlapping histograms from the three sets.

3.6 Summary and Discussion

In this chapter, we have extended previous results and made improvements to the mean-field fluid model of mixed multi-class networks of PS and INF queues with PH distributed service times, for the purpose of improving performance modeling in cloud systems. In particular, we have for these types of networks

- derived a compact matrix-form for the mean-field fluid model, and shown convergence according to Kurtz's theorem,
- introduced a computationally cheap improvement for the mean-field fluid model based on data-driven smoothing of the drift function, and
- obtained a closed-form approximation of the response time CDF over an arbitrary subset of classes in the network.

These contributions were later evaluated on two small and one larger simulation experiment with promising results.

Discussion. As shown in Section 3.2, that Kurtz's theorem should hold for mixed PS networks is quite intuitive, as it has previously been proven for closed PS networks. It is nevertheless a useful result, and there is an innovation in the compact matrix form presented in Theorem 3.1 that provides an intuitive and easily constructed mean-field fluid model to reason about the system. This matrix form should be generalizable to any form of queueing network that can be expressed as a density-dependent population process.

The smoothed mean-field fluid model introduced in Section 3.3 shows promising improvements to the standard model for systems of lower system size, and parallels can be drawn to the PSFFA method for single queues. Compared to other methods, it is simple and computationally cheap, but not as general. We only derive it for networks of PS and INF queues, but it could potentially be extended to single-class FCFS and discriminatory PS with minor effort. Another downside is that the smoothed model needs data to fit $\boldsymbol{\eta}$, but as the standard mean-field fluid model can be seen as a special case of the smoothed model without uncertainty in $\boldsymbol{X}$, simply putting $\boldsymbol{\eta}$ to some upper conservative value should in general outperform the standard model. From a performance modeling perspective, data from a running system is not an issue if the model is used for online decision making in e.g. resource management or scheduling.

The closed-form approximation of the response time CDF from Section 3.4 yields a practical and quick-to-evaluate method to retrieve percentiles. For the systems considered, the approximation seems to be accurate, but this depends on the assumption that every request receives the mean processor share. For certain systems, the assumption could hold for an adequate

approximation of the CDF at transient times, as the smoothed mean-field fluid model gives us an approximation to $\mathbb{E}\left[g\left(\boldsymbol{X}(t)\right)\right]\ \forall t \geq 0$.

The simulation results in Section 3.5 promisingly suggest that the smoothed model is robust in predicting mean queue lengths under perturbations common in cloud computing. It might seem odd that we have only considered the accuracy of stationary solutions in the randomized simulation experiment, but as can be seen in Figure 3.4, 3.6 and 3.10 the transients of the mean-field fluid models appear to be well-behaved. The transient values are thus likely to follow the same accuracy pattern as the stationary results. Another potential critique is that we have only considered idealized simulated systems, and in reality no computing system will behave exactly like a network of PS queues. However, both the smoothed model and the closed-form CDF approximation are approximations of metrics in a mixed PS queueing network. Therefore we wanted to evaluate the accuracy in such an idealized environment without worrying about unrelated modeling errors between a more realistic system and the queueing network itself. Furthermore, in Chapter 4 the smoothed mean-field model will be used to estimate performance metrics for a microservice application.

4

Runtime Extraction of a Microservice Fluid Model

Dynamic resource management is a difficult problem in modern microservice applications. Many proposed methods rely on the availability of an analytical performance model, often based on queueing theory. Such models can always be handcrafted, but this takes time and requires expert knowledge. Various methods have been proposed that can automatically extract models from logs or tracing data. However, these methods are often intricate, requiring offline stages and advanced algorithms for retrieving the service time distributions. Although accurate, the resulting models can be complex and unsuitable for online evaluation. Thus, when designing a method for automatic model extraction for runtime management, apart from accuracy, one needs to take into account both the extraction complexity and the method of evaluation for the resulting queueing model.

Introduction

In this chapter, we build upon the results from Chapter 3 to propose a fluid model for microservice applications, that can be extracted at runtime in a distributed fashion from common local tracing data at each service replica. The model is derived as the *smoothed mean-field fluid model* of a simple yet general queueing network model of the application. In this intermediate queueing network, each replica is represented as a single multi-class PS queue and each service-to-service transition as an INF queue, and each class is assumed to have a PH-distributed service time. The queueing network is further allowed to be *mixed*, which implies that it can model applications subjected to Poisson arrivals and/or a set of connecting clients.

The resulting model and its flexibility are showcased on a multi-cluster microservice application. However, as it is in general cumbersome to setup the necessary environment to test such applications, we developed a sandbox to streamline this effort by emulating a multi-cloud environment on an IaaS

Table 4.1 Local nomenclature for Chapter 4

Notation	Description
	Service Graph
$\mathcal{M}$	Set of services (instances).
$\mathcal{N}_m$	Set of request types in service $m \in \mathcal{M}$.
k_m	Number of CPU cores in service m (considered known).
$\mathcal{U}_{m,n}$	Set of all upstream destinations in $(m,n) \in (\mathcal{M}, \mathcal{N}_m)$.
$\mathcal{D}_{m,n}$	Set of all downstream sources in (m,n).
$\mathcal{E}_o$	Set of (m,n) under an open connection.
$\mathcal{E}_c$	Set of (m,n) under a closed connection.
	Tracing Data (considered known)
$\mathcal{R}_{m,n}$	Set of requests in (m,n).
$\boldsymbol{U}^r$	Vector of upstream destinations for $r \in \mathcal{R}_{m,n}$.
D^r	Downstream source for r.
t^r	Arrival timestamp for r.
Δt^r	Duration for r.
$\boldsymbol{T}^r$	Vector of timestamps for the remote calls of r.
$\Delta \boldsymbol{T}^r$	Vector of durations for the remote calls of r.
$\Delta \overline{\boldsymbol{T}}^r$	Vector of upstream durations for the remote calls of r.

platform. The microservice fluid model is found to accurately capture important performance metrics of the application deployed using this sandbox, both for the current operating conditions and for predictions under common system perturbations.

Outline. This chapter is structured as follows. In the end of this introduction the necessary notation and assumptions on the microservice application, along a quick recap of the smoothed mean-field fluid model, are given. Section 4.1 then introduces the queueing network model and explains the distributed extraction from local tracing data. Furthermore, Section 4.2 shows how the model can be adapted to perform predictions under common system perturbations. In Section 4.3 the multi-cloud sandbox is explained in more detail, and later used to perform experimental evaluations in Section 4.4 to test the accuracy of both model and predictions. Finally, the chapter is summarized in Section 4.5.

Assumptions and Notations

For an overview of the introduced notation, see Table 4.1. First, in this chapter (a,b) or $[a,b]$ will denote a tuple of a and b, while $a:b$ denotes the ordered set $\{k \in \mathbb{Z} \; s.t. \; a \leq k \leq b\}$. Writing $(a,b) \in [\mathcal{A}, \mathcal{B}_a]$ implies $b \in \mathcal{B}_a$, where $a \in \mathcal{A}$. If these (a,b) can partition some set $\mathcal{C}$, they can be used to denote a subset as $\mathcal{C}_{a,b} \subset \mathcal{C}$. Leaving out b implies a union, that is, $\mathcal{C}_a = \bigcup_{b \in \mathcal{B}_a} \mathcal{C}_{a,b}$.

Microservice application. We assume an application consisting of a set of microservices that interact in a graph structure. Each service can have multiple instances, i.e., replicas. Let $\mathcal{M}$ be the set of all instances of all services, and let "service" henceforth in this chapter be synonymous with "service instance" if applicable. The services are assumed to run in mutually isolated environments, and in a slight abuse of notation we let k_m be the number of CPU cores per service m and considered known.

Each service is assumed to be able to simultaneously handle multiple requests of different types. These types are assumed distinct to each service, hence the execution of an external request can visit multiple request types in the service graph. We define $\mathcal{N}_m$ as the set of request types that service m can handle. Processing a request of type $n \in \mathcal{N}_m$ is assumed to consist of (i) an arrival, (ii) a set of sequentially executed remote calls to request types in upstream services, (iii) a response to the downstream caller upon completion, and (iv) processing times in service m between the arrival, the remote calls and the response. We assume that all remote calls are *synchronous*, meaning that no processing is performed in the service on the request issuing the call before a response is returned. Furthermore, a remote call is always made to a single upstream service, disallowing fork–join behavior. The set of request types could represent different endpoints in service m, but exactly how the requests in m are partitioned into types is not important. What is important is that all requests in each request type $n \in \mathcal{N}_m$ have the same number of remote calls. Let a specific request type n in the service m be expressed as the pair $(m, n) \in (\mathcal{M}, \mathcal{N}_m)$. Furthermore, let $\mathcal{U}_{m,n} := \{(\overline{m}_1, \overline{n}_1), \dots\}$ be the set of all upstream request types in services that receive remote calls from (m, n), and let $\mathcal{D}_{m,n} := \{(\underline{m}_1, \underline{n}_1), \dots\}$ be the set of all downstream request types in services that make remote calls to (m, n). We assume that multiple downstreams to (m, n) exist only as replicas of the same microservice.

Requests can arrive to (m, n) from external sources, either through an *open connection* as Poisson arrivals, or through a *closed connection* as a set of external clients utilizing the application. Once a client request has been returned, the client waits some time before submitting another request. Let $\mathcal{E}_o$ be a set of all (m, n) under an open connection, and $\mathcal{E}_c$ be a set of all (m, n) under a closed connection.

Let $\mathcal{R}_{m,n}$ be the set of requests of type n to visit the service m in some time window. For each $r \in \mathcal{R}_{m,n}$, let $\boldsymbol{U}^r$ be a vector of upstream destinations for each remote call ordered by their sequential execution, such that each element is a pair $(\overline{m}, \overline{n})$ of the upstream service and request type. Since remote calls might be load balanced on different replicas, $\boldsymbol{U}^r \subset \mathcal{U}_{m,n}\ \forall r \in \mathcal{R}_{m,n}$. The number of remote calls $|\boldsymbol{U}^r|$ will be the same for all requests visiting (m, n). Furthermore, let D^r be the downstream source of the request. If the downstream is part of the service graph, then $D^r = (\underline{m}, \underline{n})$. Otherwise, $D^r = D^r_c$ for a closed connection, while $D^r = D^r_o$ for an open connection. We

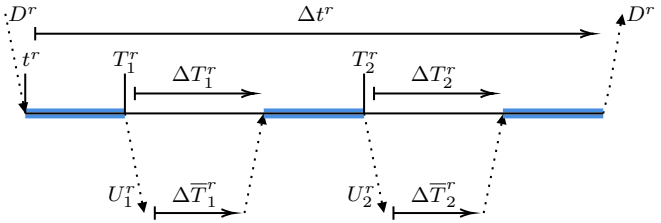


Figure 4.1 The local trace of request $r \in \mathcal{R}_{m,n}$ of request type n in some service m, performing two remote calls. This implies that all requests in $\mathcal{R}_{m,n}$ also performs two remote calls before returning to the downstream.

assume that it is possible to differentiate D_o^r from D_c^r, and furthermore that the ID of D_c^r is unique for each connecting client.

We define the *local request trace* of request r in (m, n) as (i) a time of arrival t^r, (ii) a duration Δt^r, and (iii) three vectors $\boldsymbol{T}^r, \Delta\boldsymbol{T}^r$ and $\Delta\overline{\boldsymbol{T}}^r$ containing the timestamp/duration/upstream duration of remote call i. See Figure 4.1 for an illustration. Let $\mathcal{H}^r := \left\{D^r, \boldsymbol{U}^r, t^r, \Delta t^r, \boldsymbol{T}^r, \Delta\boldsymbol{T}^r, \Delta\overline{\boldsymbol{T}}^r\right\}$ be the local tracing data for request r. We assume that, locally, $\mathcal{H}_r$ is known $\forall r \in \mathcal{R}_m$ in all services m. These local request traces can commonly be captured in software that supports distributed tracing. For example, see the documentation on the Envoy proxy regarding access logging[1].

Recapping the smoothed mean-field fluid model. We adopt similar queueing network assumptions and notations to that of Chapter 3. Let $\mathcal{Q}$ define the set of queues in the network, where each $q \in \mathcal{Q}$ has a set of classes $\mathcal{C}_q$. Each class c in each queue q further has a PH-distributed service time with $\mathcal{S}_{q,c}$ as its set of phases, where $\psi^{q,c} \in \mathbb{R}^{|\mathcal{S}_{q,c}|\times 1}$, $\Psi^{q,c} \in \mathbb{R}^{|\mathcal{S}_{q,c}|\times|\mathcal{S}_{q,c}|}$ and $\zeta^{q,c} \in \mathbb{R}^{|\mathcal{S}_{q,c}|\times 1}$ are its three parametrization matrices. Let $\boldsymbol{X} \in \mathbb{Z}_+^{|\mathcal{S}|\times 1}$ be the population of requests in each phase state. Each class can potentially have Poisson arrivals with rates described by $\boldsymbol{\lambda} \in \mathbb{R}^{|\mathcal{C}|\times 1}$. Furthermore, let $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$ be the routing probability matrix between all classes. Finally, each queue q is assumed to follow either an INF or a PS discipline, and for the PS queues it is assumed that the number of servers is the same as the number of cores k_m in the service it models.

The time evolution of $\boldsymbol{X}$ can be described as a continuous-time Markov chain, and using the smoothed mean-field fluid model from Definition 3.2

[1] https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/access_log/usage

$\mathbb{E}[\boldsymbol{X}(t)]$ can be approximated with the ODE

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= \boldsymbol{W}^T \hat{\theta}(\boldsymbol{x}, \boldsymbol{\eta}) + \boldsymbol{A}\boldsymbol{\lambda}, \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0), \quad \boldsymbol{\eta}(t) > \boldsymbol{0}, \end{aligned} \tag{4.1}$$

where $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$, $\hat{\theta}_i(\boldsymbol{x}, \boldsymbol{\eta}) = x_i \hat{g}_{Q(i)}\left(\boldsymbol{x}, \eta_{Q(i)}\right)$, and $\hat{g}_q(\boldsymbol{x}, \eta_q)$ is defined as (3.31) for some phase state i and queue q. The matrices $\boldsymbol{\Psi}, \boldsymbol{B}, \boldsymbol{A}$ are block diagonals of the PH distribution matrices stacked in the appropriate order. Let $\boldsymbol{X}^*$ and $\boldsymbol{x}^*$ denote the stationary distribution of the phase-state populations and the stationary solution of (4.1), respectively. From the mean queue length and utilization ρ of a stationary system, it is always possible to find an optimal $\boldsymbol{\eta}$ for the current operating conditions by solving (3.35). Furthermore, the response time CDF can be approximated via (3.48).

4.1 Capturing the Model

In this section, we will show how our rudimentary queueing network model is constructed from a microservice application and how it can be obtained given the defined tracing data. Later, we show that the model can be extracted in a distributed fashion and discuss how this may be implemented in a real system.

From Service Graph to Queueing Network Model

In the queueing network model, we represent each service m as single multi-class PS queues and the interservice delays and external clients by multi-class INF queues. An illustration of the translation is shown in Figure 4.2. Here, an example application consisting of 3 microservices (blue, orange and green) is considered and its corresponding service graph shown in Figure 4.2(a). Green has two replicas, while the others have one, yielding $\mathcal{M} = \{m_1, m_2, m_3, m_4\}$. Requests visiting either orange or blue can be partitioned into two request types, while requests visiting green has a single type, i.e., $\mathcal{N}_{m_1} = \{n_1, n_2\}$, $\mathcal{N}_{m_2} = \{n_1\}$, $\mathcal{N}_{m_3} = \{n_1\}$ and $\mathcal{N}_{m_4} = \{n_1, n_2\}$. External requests arrive to m_1 either to request type n_1 in a closed connection, or to n_2 in an open connection. (m_1, n_1) has one remote call to (m_4, n_1), and (m_1, n_2) has one remote call to either (m_2, n_1) or (m_3, n_1) decided by some load balancing strategy. They in turn have one remote call to (m_4, n_2). As the single remote call in (m_1, n_2) has two potential upstreams, $\mathcal{U}_{m_1,n_2} := \{(m_2, n_1), (m_3, n_1)\}$ while $\boldsymbol{U}^r$ is *either* (m_2, n_1) or (m_3, n_1) for every $r \in \mathcal{R}_{m_1,n_2}$. Equivalently, as (m_4, n_2) is called by two downstreams, $\mathcal{D}_{m_4,n_2} := \{(m_2, n_1), (m_3, n_1)\}$ while D^r is either (m_2, n_1) or (m_3, n_1) for every $r \in \mathcal{R}_{m_4,n_2}$.

How to translate such an application to the queueing network model will be explained in more detail in this subsection.

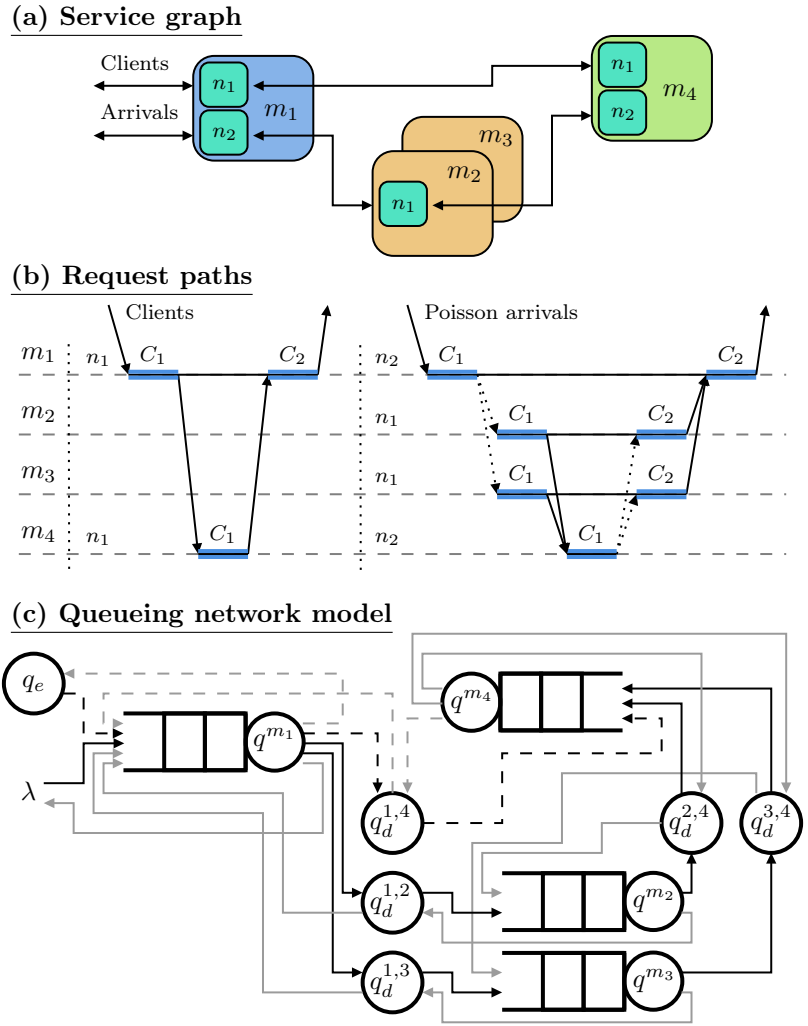


Figure 4.2 The translation from (a) a microservice application graph, to (c) the corresponding queueing network model. In (b), the paths that all external requests to either request type in m_1 must traverse, along the corresponding class decomposition, are shown. Here, the dotted arrows shows where classes have multiple outgoing connections. In (c), the services have been replaced by PS queues, the service-to-service connection delays by INF queues, and the external client waiting times with an INF queue q_e. Each arrow shows a class-to-class transition, where the dashed arrows are part of the corresponding closed chain, and the solid part of the corresponding open chain. The grey arrows shows the returning transitions.

Service queues. Let q^m be the PS queue that models the service $m \in \mathcal{M}$. Due to the fixed number of sequential remote calls for each $r \in \mathcal{R}_{m,n}$, in each (m,n), r will be processed in $|\boldsymbol{U}^r| + 1$ disjoint steps: first after arrival, and then after each remote call. After all processing steps are completed, the request is returned to its downstream caller $(\underline{m}, \underline{n})$. Furthermore, while making a remote call, it is assumed that the processing of r pauses in (m,n) until the call is completed. Thus, the request r can be considered to "depart" from the current processing step for the upstream $(\overline{m}, \overline{n})$ and rejoin the next processing step once the remote call is completed.

This behavior can in turn be captured in a queueing network by letting the time of each processing step be modeled as the service time in a standalone class for a total of $|\mathcal{C}_{q^m,n}| = |\boldsymbol{U}^r| + 1$ classes for requests visiting (m,n). We assume that all requests arriving at any (m,n) enter the first class and depart from the final class of $\mathcal{C}_{q^m,n}$, and that all remote calls from class c are returned to $c{+}1$. This is illustrated in Figure 4.2(a) and 4.2(b), where the single remote calls in (m_1,n_1), (m_1,n_2), (m_2,n_1), (m_3,n_1) give rise to two classes in each service/request type. Here, requests are assumed to always enter the first class C_1 and depart the final class, i.e., C_2 or C_1, when completed.

Delay and client queues. As services can be hosted far apart in the cloud, there might be nonnegligible delays between services that need to be taken into account. For each remote call between two services $m \to \overline{m}$, we introduce an INF queue $q_d^{m,\overline{m}}$ to model the transfer delay. For each pair of request types $(n, \overline{n}) \in (\mathcal{N}_m, \mathcal{N}_{\overline{m}})$ in such a remote call, we introduce two classes in $q_d^{m,\overline{m}}$ and let the first class handle the delay and routing for $(m,n) \to (\overline{m}, \overline{n})$ and the second class $(\overline{m}, \overline{n}) \to (m,n)$. Thus, we get $|\mathcal{C}_{q_d^{m,\overline{m}}}| = 2|\overline{\mathcal{N}}_m|$ where $\overline{\mathcal{N}}_m$ is the set of all upstream request types called from the service m. Furthermore, if $\mathcal{E}_c$ is not empty, we introduce a new INF queue q_e with $|\mathcal{C}_{e^c}| = |\mathcal{E}_c|$. Each class $c \in \mathcal{C}_{e^c}$ models the client waiting time and handles the routing to an element $(m,n) \in \mathcal{E}_c$, which after completion at (m,n) routes back to c again. Returning to Figure 4.2, (c) shows the complete queueing model of the example application with the delay queues $q_d^{1,2}, q_d^{1,3}, q_d^{1,4}$ for each remote call from m_1 and $q_d^{2,4}, q_d^{3,4}$ for the remote calls from m_2 and m_3, along with the client waiting queue q_e.

Properties of the routing probability matrix. In the queueing model, most classes will have a single outgoing connection. This can in turn be captured in the routing probability matrix $\boldsymbol{P}$ as static routing, i.e., classes with a single outgoing connection will have a single nonzero element with value 1 in its corresponding row in $\boldsymbol{P}$. The only two cases where classes have more than one outgoing connection are (i) when a remote call has multiple possible upstreams, and (ii) when a request type receives from multiple downstreams.

In the first case, the class right before the remote call will have a connec-

tion to the first class in each of the possible upstreams. This can be captured in $\boldsymbol{P}$ by assuming that each request is assigned to its upstream at random, e.g., via a load balancer that follows a weighted random policy. Each class that perform random routing to multiple classes will thus have multiple routing probabilities in its corresponding row in $\boldsymbol{P}$.

In the second case, the final class of (m, n) must return responses to the correct next class in each downstream. This can be captured in $\boldsymbol{P}$ for the fluid model by using random routing with probabilities chosen such that the flow back to the downstream classes is preserved. As we only allow multiple downstreams as replicas to the same microservice, effects such as flow spilling over between external sources and shortcuts for requests when calculating the response time percentiles are mitigated. When estimating parameters from data these flow preserving probabilities will be obtained automatically, but care should be taken for predictions, as discussed in Section 4.2.

Finally, a class can potentially also have no outgoing connections, which happens at the final class of (m, n) when the first class is under Poisson arrivals. This can be captured in $\boldsymbol{P}$ as an empty row.

For example, the resulting queueing network for the example application shown in Figure 4.2(c) has only single-valued rows in $\boldsymbol{P}$ except for (i) C_1 in (m_1, n_2) which has two nonzero values for routing to C_1 in (m_2, n_1) and (m_3, n_1), (ii) C_1 in (m_4, n_2) which has two nonzero values for preserving flow to C_2 in (m_2, n_1) and (m_3, n_1), and (iii) for C_2 in (m_1, n_2) which will have only zero values for departing requests.

The following properties of the paths over the classes created by $\boldsymbol{P}$ can be stated, which is of importance for the response time CDF approximation.

Remark 4.1
The class-to-class path formed by each open connection is an open chain, and each closed connection a closed chain. Also,

- the open chains are *transient*, i.e., starting from any class in the chain, there is a nonzero probability of leaving the network in finite time.
- the closed chains are *irreducible*, i.e., starting from any class in the chain, there is a nonzero probability of visiting any other class in the chain in finite time.

This holds by construction. Due to the absence of multiple downstreams (except for replicas of the same microservice), requests from one external connection will never reach the same (m, n) as another external connection. Further, from the first class visited by an external request, there trivially exists a path to any other class that the execution of the external request can visit. From any of those classes, there exists a path to the final class before departure. For open connections, the request departs the network while for

closed connections the request is returned to the first class after some time. In Figure 4.2, the open and closed connections give rise to one open chain and one closed chain, symbolized by the dashed and solid arrows respectively in (c).

Queue and class partitions. The set of queues can be partitioned as $\mathcal{Q} = \mathcal{Q}_M \cup \mathcal{Q}_d \cup \{q_e\}$ where

$$\begin{aligned} \mathcal{Q}_M &:= \{q^m \ \forall m \in \mathcal{M}\}, \\ \mathcal{Q}_d &:= \left\{q_d^{m,\overline{m}} \ \forall (m,\overline{m}) \in (\mathcal{M},\mathcal{U}_m)\right\}. \end{aligned} \tag{4.2}$$

Furthermore, we can—given an arbitrary class c in the set of all classes $\mathcal{C}$ for which q is its parent queue—partition and express c as the following tuple depending on the type of q:

$$c = \begin{cases} [q^m, n, u] \in [\mathcal{Q}_M, \mathcal{N}_m, \mathcal{C}_{q^m,n}] & q \in \mathcal{Q}_M, \\ \left[q_d^{m,\overline{m}}, (n,\overline{n}), u\right] \in \left[\mathcal{Q}_d, \left(\mathcal{N}_m, \overline{\mathcal{N}}_m\right), \{C_1, C_2\}\right] & q \in \mathcal{Q}_d, \\ [q_e, (m,n), u] \in [\mathcal{Q}_e, (\mathcal{M}, \mathcal{N}_m), \{C_1\}] & q = q_e. \end{cases} \tag{4.3}$$

This allows us to keep track of the request type and queue that each class in $\mathcal{C}$ belongs to, which will be important for obtaining the network parameters. For example, considering the example application in Figure 4.2, class C_1 in (m_1, n_1) can be expressed as $[q^{m_1}, n_1, C_1]$ and C_2 in $q_d^{2,4}$ as $\left[q_d^{2,4}, (n_1, n_2), C_2\right]$.

Estimating Network Parameters

Here, we will show how the introduced queueing network model can be extracted from the tracing data. Assume that we have $\mathcal{H}^r$ collected for all $r \in \mathcal{R}$ over all services in a system at stationarity, over some time period $\Delta T_{\mathcal{R}}$.

Extracting routing probabilities. The routing between services and the influx of external requests must be transformed into class-to-class transition rates to form $\boldsymbol{P}$ and $\boldsymbol{\lambda}$.

First, we introduce $w^o \in \mathbb{Z}_+^{|\mathcal{C}|\times 1}$ to denote the number of requests in $\mathcal{R}$ that arrive to the classes from the open connections. Each element in w^o is initially set to 0. For each $r \in \mathcal{R}$ where $D^r = D_o^r$, we then identify (m,n) and increment $w^o_{[q^m,n,C_1]}$ by 1. The arrival rates can then be estimated as

$$\lambda_c \approx \frac{w_c^o}{\Delta T_{\mathcal{R}}} \qquad \forall c \in \mathcal{C}, \tag{4.4}$$

implying that $\lambda_c \geq 0$ only if $c \in [\mathcal{Q}_M, \mathcal{N}_m, C_1]$ and $(m,n) \in \mathcal{E}_o$.

For closed connections and routing between classes within the network, we introduce $w \in \mathbb{Z}_{+}^{|\mathcal{C}|\times|\mathcal{C}|}$ to denote the number of requests in $\mathcal{R}$ that initiate a connection between any two classes $a, b \in \mathcal{C}$. Each element in w is initially set to 0. For the closed connection, for each $r \in \mathcal{R}$ where $D^r = D^r_c$, we identify (m, n) and increment $w_{a,b}$ by 1 for each

$$(a,b) \in \Big\{ \Big([q_e, (m,n), C_1], [q^m, n, C_1] \Big), \\ \Big([q^m, n, C_f], [q_e, (m,n), C_1] \Big) \Big\}, \tag{4.5}$$

where C_f is the final class of $\mathcal{C}_{q^m,n}$. I.e., for every request from an external client, we identify its (m, n) and the correct class partitions for the transitions (i) from client to (m, n), and (ii) returning from (m, n) to client. For both of these class tuples, $w_{a,b}$ is incremented by 1.

Furthermore, for the routing between classes inside the service graph, for each $r \in \mathcal{R}$ we identify (m, n), and for each $u \in 1 : |\boldsymbol{U}^r|$ where $U^r_u = (\overline{m}, \overline{n})$, we increment $w_{a,b}$ by 1 for each

$$(a,b) \in \Big\{ \Big([q^m, n, C_u], \Big[q_d^{m,\overline{m}}, (n,\overline{n}), C_1\Big] \Big), \\ \Big(\Big[q_d^{m,\overline{m}}, (n,\overline{n}), C_1\Big], \big[q^{\overline{m}}, \overline{n}, C_1\big] \Big), \\ \Big(\big[q^{\overline{m}}, \overline{n}, C_f\big], \Big[q_d^{m,\overline{m}}, (n,\overline{n}), C_2\Big] \Big), \\ \Big(\Big[q_d^{m,\overline{m}}, (n,\overline{n}), C_2\Big], [q^m, n, C_{u+1}] \Big) \Big\}, \tag{4.6}$$

where C_f is the final class of $\mathcal{C}_{q^{\overline{m},\overline{n}}}$. I.e., for every outgoing remote call in every request, we identify its (m, n) and the correct class partitions for the transitions (i) from (m, n) to the first class of the corresponding delay queue, (ii) from the delay queue to $(\overline{m}, \overline{n})$, (iii) returning from $(\overline{m}, \overline{n})$ to the second class of the delay queue, and (iv) from the delay queue to the next processing step in (m, n). For all four class tuples, $w_{a,b}$ is incremented by 1.

Finally, the elements of the routing probability matrix can then be estimated as

$$\boldsymbol{P}_{a,b} \approx \frac{w_{a,b}}{\sum_{c\in\mathcal{C}} w_{a,c}}, \tag{4.7}$$

for all permutations of $a, b \in \mathcal{C}$. In the case that $\sum_{c\in\mathcal{C}} w_{a,c} = 0$, all elements in the corresponding row $\boldsymbol{P}_{a,\cdot}$ is set to 0.

Extracting service time distributions. To estimate the service time distribution for each class c, we first obtain the arrival/departure times (t^a, t^d)

of all requests visiting c. We do this by considering the queues in $\mathcal{Q}_M$, $\mathcal{Q}_d$ and q_e separately. First, let $\mathcal{T}_c := \left\{\left(t^a_{c,r}, t^d_{c,r}\right)\ r \in \mathcal{R}\right\}$ be the set of all pairs of arrival/departure times for requests that visit c.

Taking into account the type of request n in the service m, we can transform all timestamps/durations for each $r \in \mathcal{R}_{m,n}$ and its remote calls to arrival/departure times for each $c \in [q^m, n, \mathcal{C}_{q^m,n}]$. Let $u \in 1 : |\mathcal{C}_{q^m,n}|$, then if $|\mathcal{C}_{q^m,n}| > 1$

$$\mathcal{T}_c = \begin{cases} \{(t^r, T^r_1)\} & u = 1, \\ \left\{\left(T^r_{u-1} + \Delta T^r_{u-1}, t^r + \Delta t^r\right)\right\} & u = |\mathcal{C}_{q^m,n}|, \\ \left\{\left(T^r_{u-1} + \Delta T^r_{u-1}, T^r_u\right)\right\} & \text{otherwise}, \end{cases} \tag{4.8}$$

else $\mathcal{T}_c = \left\{(t^r, t^r + \Delta t^r)\right\}, \forall r \in \mathcal{R}_{m,n}$. This can be clearly seen by considering Figure 4.1.

Considering the delay queues, there are no direct timestamps/durations available as they do not correspond to any measured services. Instead, for each $r \in \mathcal{R}_{m,n}$ and for each $u \in 1 : |\boldsymbol{U}^r|$ where $U^r_u = (\overline{m}, \overline{n})$, we can estimate the delay from the upstream duration $\Delta\overline{T}^r_u$ assuming equality in both directions, i.e., $\tau^r_u = \left(\Delta T^r_u - \Delta\overline{T}^r_u\right)/2$, and find

$$\begin{aligned} \mathcal{T}_c &= \left\{(T^r_u, T^r_u + \tau^r_u)\right\}_{r \in \mathcal{R}_{m,n}}, \\ \mathcal{T}_{c'} &= \left\{\left(T^r_u + \Delta\overline{T}^r_u + \tau^r_u, T^r_u + \Delta T^r_u\right)\right\}_{r \in \mathcal{R}_{m,n}}, \end{aligned} \tag{4.9}$$

where $c = \left[q^{m,\overline{m}}_d, (n, \overline{n}), C_1\right]$, $c' = \left[q^{m,\overline{m}}_d, (n, \overline{n}), C_2\right]$.

Finally, considering the client waiting time queue q_e, we also here have neither direct timestamp/duration measurements of requests visiting its classes. However, as we know that a departing request of request type n in service m to $[q_e, (m, n), C_1]$ will return to (m, n) once the waiting time has passed, we can obtain the arrivals/departures by finding which $\overline{r} \in \mathcal{R}_{m,n}$ is the next arrival to (m, n) of the client that sent $r \in \mathcal{R}_{m,n}$. As we assume that $D^r = D^r_c$ constitutes a unique ID for each connecting client for r when $(m, n) \in \mathcal{E}_c$, we can thus for each $c = [q_e, (m, n), C_1]$ find the arrival/departures as

$$\begin{aligned} \mathcal{T}_c &= \left\{\left(t^r + \Delta t^r, t^{\overline{r}}\right)\right\}_{r \in \mathcal{R}_{m,n}}, \\ \text{where} \quad \overline{r} &= \underset{\substack{\overline{r} \in \mathcal{R}^{m,n} \\ D^{\overline{r}} = D^r}}{\arg\min} \left(t^{\overline{r}} > t^r + \Delta t^r\right). \end{aligned} \tag{4.10}$$

Repeating (4.8), (4.9), and (4.10) for all (m, n), we can retrieve the complete dataset of arrivals/departures for every class in the queueing network under $\mathcal{R}$.

For any $c \in \mathcal{C}$, the class population can then be calculated at any time t as the difference between the number of arrivals and departures

$$X_c(t) = \sum_{\left(t^a_{c,r}, t^d_{c,r}\right) \in \mathcal{T}_c} \mathbb{1}\left[t > t^a_{c,r}\right] - \mathbb{1}\left[t > t^d_{c,r}\right]. \tag{4.11}$$

The total queue length in any $q \in \mathcal{Q}$ is further given by

$$X_q(t) = \sum_{c \in \mathcal{C}_q} X_c(t). \tag{4.12}$$

For a given $c \in \mathcal{C}$, if the corresponding queue q follows an INF discipline, i.e., $q \in \mathcal{Q}_d \cup \{q_e\}$, then the service times can be obtained directly as

$$\mathcal{T}^s_c := \left\{t^d_{c,r} - t^a_{c,r}\right\}_{\left(t^a_{c,r}, t^d_{c,r}\right) \in \mathcal{T}_c}. \tag{4.13}$$

If the class instead belongs to a PS queue, extracting the service times becomes more complicated. Consider $r \in \mathcal{R}_{m,n}$, and let $\boldsymbol{V}^r$ be the sorted vector of all arrival/departure times to any class in $\mathcal{C}_{q^m}$ that lies within the bound $[t^a_{c,r}, t^d_{c,r}]$, that is,

$$\boldsymbol{V}^r = \texttt{sort}\left(\left\{t_m \in \mathcal{T}_m \; s.t. \; t^a_{c,r} \leq t_m \leq t^d_{c,r}\right\}\right), \tag{4.14}$$

where $\mathcal{T}_m$ is the union of all arrival and departure times in service m. The service time $t^s_{c,r}$ can then be obtained as [Perez et al., 2015]

$$t^s_{c,r} = \sum_{i=1}^{|\boldsymbol{V}^r|-1} \frac{\left(V^r_{i+1} - V^r_i\right) k_m}{\max\left[k_m, X_{q^m}\left(V^r_i\right)\right]}, \tag{4.15}$$

and hence $\mathcal{T}^s_c = \{t^s_{c,r}\}_{\left(t^a_{c,r}, t^d_{c,r}\right) \in \mathcal{T}_c}$. From the sets of service times $\mathcal{T}^s_c$ it is then straightforward to fit a PH distribution and obtain Ψ^c, ψ^c, α^c using, e.g., moment matching [Osogami and Harchol-Balter, 2006] or the EM algorithm [Asmussen et al., 1996], as long as there is enough data to capture the moments of the underlying distributions.

Extracting smoothing parameters. Good values for the smoothing parameters can be obtained by using (3.35) together with estimated utilizations for each $q^m \in \mathcal{Q}_M$ over $\mathcal{R}$ using either the utilization law or an empirical distribution of the queue length probabilities.

Distributed Tracking

So far, we have assumed that all data is available at some central observer. This is possible in theory, but not feasible in practice, as the aggregation of

trace data $\mathcal{H}^r$ for each request from each service can create a large overhead in the cluster. In setups for distributed tracing, overhead is often reduced by only aggregating data such as $\mathcal{H}_r$ on a small subset of all requests. This is a problem for our model, as all samples are needed to extract the service time distributions.

However, if we assume that we can gather the necessary data for each request and store them locally at each service m, then it is possible to extract the model in a distributed fashion. Assume that in each m we know $\mathcal{H}_m := \{\mathcal{H}^r\}_{r\in\mathcal{R}_m}$, then:

THEOREM 4.1
Given $\mathcal{H}_m$ for $m \in \mathcal{M}$. Then $\boldsymbol{P}_{c,\cdot}$, $(\Psi^c, \psi^c, \zeta^c)$ and λ_c can be calculated for all classes in q^m, its upstream delays $q_d^{m,\overline{m}}$ and associated downstream classes in q_e, i.e., $\forall c \in \mathcal{C}_{q^m} \cup \left(\bigcup_{\overline{m}\in\mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}}\right) \cup \mathcal{C}_{q_e,m}$ where $\mathcal{C}_{q_e,m} := \{[q_e, (m', n'), C_1] \in \mathcal{C}_{q_e} \; s.t. \; m' = m\}$.

Proof. For λ_c, $\mathcal{H}_m$ contains D^r $\forall r \in \mathcal{R}_m$, therefore, from (4.4), we see that all the information needed to estimate Poisson rates $\forall c \in [q^m, \mathcal{N}_m, C_1]$ is available.

For $\boldsymbol{P}_{c,\cdot}$, $\mathcal{H}_m$ also contains $\boldsymbol{U}^r$ $\forall r \in \mathcal{R}_m$, therefore, from (4.6) and (4.7) we see that the information is available to estimate all outgoing routing probabilities from $c \in \mathcal{C}_{q^m} \cup \left(\bigcup_{\overline{m}\in\mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}}\right)$. Similarly, for $c \in \mathcal{C}_{q_e,m}$, from (4.5) we see that D^r $\forall r \in \mathcal{R}_m$ is needed which is included in $\mathcal{H}_m$.

Finally, $(\Psi^c, \psi^c, \zeta^c)$ can be estimated from $\mathcal{T}_c^s$. From (4.13) we see that we only need $\mathcal{T}_c$ if $c \in \left(\bigcup_{\overline{m}\in\mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}}\right) \cup \mathcal{C}_{q_e,m}$, which from (4.9) are obtainable for upstream delays and from (4.10) for the associated classes in the client queue given only $\mathcal{H}_m$. If instead we consider the classes in q^m, from (4.15) and (4.14) we need $\mathcal{T}_c$ $\forall c \in \mathcal{C}_{q^m}$, k_m, and X_{q^m}. However, k_m is considered known, and from (4.8) and (4.12) the other two values can be obtained by considering only $\mathcal{H}_m$. □

Therefore, we can obtain local queueing network model parameters for each service, using only the local tracing data. When all the parameters are combined, the local parameters form the entire model as shown below.

COROLLARY 4.1.1
Aggregating the local model parameters $\forall m \in \mathcal{M}$ yields the entire queueing network model, i.e.,

$$\bigcup_{m\in\mathcal{M}} \mathcal{C}_{q^m} \cup \left(\bigcup_{\overline{m}\in\mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}}\right) \cup \mathcal{C}_{q_e,m} = \mathcal{C}. \tag{4.16}$$

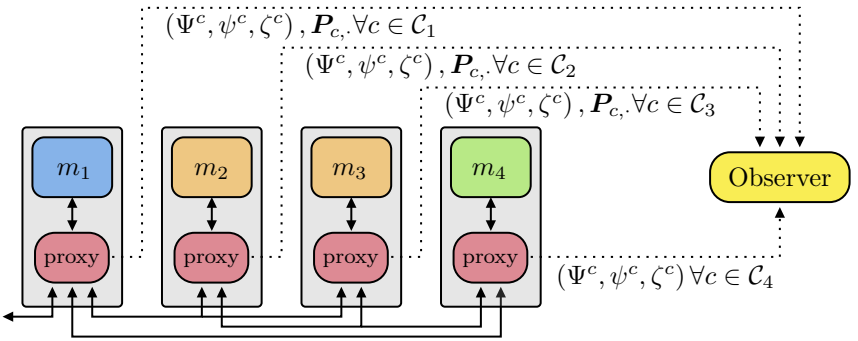


Figure 4.3 Distributed tracking of the local queueing network model parameters for the application shown in Figure 4.2. Here $\mathcal{C}_1 = \mathcal{C}_{q^{m_1}} \cup \mathcal{C}_{q_d^{1,2}} \cup \mathcal{C}_{q_d^{1,3}} \cup \mathcal{C}_{q_d^{1,4}} \cup \mathcal{C}_{q_e}$, $\mathcal{C}_2 = \mathcal{C}_{q^{m_2}} \cup \mathcal{C}_{q_d^{2,4}}$, $\mathcal{C}_3 = \mathcal{C}_{q^{m_3}} \cup \mathcal{C}_{q_d^{3,4}}$ and $\mathcal{C}_4 = \mathcal{C}_{q^{m_4}}$. From Corollary 4.1.1, $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_4 = \mathcal{C}$.

Proof. By definition, $\bigcup_{m \in \mathcal{M}} \mathcal{C}_{q^m} \cup \mathcal{C}_{q_e,m}$ covers all classes in the services and the client queue. For the delay queues, we know that every queue is created as the result of a remote call from any service m to an upstream service $\overline{m}$, and therefore $\bigcup_{m \in \mathcal{M}} \left(\bigcup_{\overline{m} \in \mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}} \right)$ must cover all classes in every delay queue. □

Finally, to estimate the smoothing parameter η_{q^m} from (3.35) we see that it only needs data from $\mathcal{H}_m$ and local utilization, which from, e.g., (2.5) only requires information in $\mathcal{H}_m$.

Feasible implementation. Theorem 4.1 allows us to create methods for the online extraction of our model, which are feasible to implement on a running microservice application. This could, for example, be achieved using a service mesh software that uses sidecar proxies, such as Istio[2]. Each service is then deployed together with a small, local proxy that handles all incoming and outgoing network traffic. These sidecar proxies thus offer a natural point to handle the extraction of local tracing data; Istio, for example, supports capturing $\mathcal{H}_m$ via its Envoy proxies almost out of the box. Thus, each sidecar proxy could be designed to log $\mathcal{H}_m$ for every request in some local buffer. This data could then be used to estimate the parameters of the local queueing network model over some sampling period, which could then be sent to the central observer. See Figure 4.3 for an illustration of such a system.

[2] https://istio.io/

4.2 Prediction Under Structural Changes

To be usable for online decision making, it is important that the model can be used to make adequate predictions of important performance metrics under perturbations to the running system. The modeled services and delays will not function as pure PS or INF queues; hence, both service times and smoothing parameters will likely be workload dependent. In Section 4.4 we will test both how well the model can capture the current operating conditions and how well it can be predicted from a previously observed condition.

First, we will present how the model can be used to predict performance metrics under workload changes, scaling, and migration. We assume that we have obtained a smoothed mean-field fluid model from some running system and want to predict what happens if we perturb the system at t^+. To make such a prediction, the parameters of the fluid model need to be updated correctly, depending on the type of perturbation. Let the superscript $+$ denote an updated parameter at time t^+.

Updating $\boldsymbol{P}$. A system perturbation could affect the inflow to a request type n in service m. If there exist multiple downstreams connecting to $c_{in} = [q^m, n, C_1]$, care must be taken to ensure that the return flow from $c_{ret} = [q^m, n, C_f]$ is correctly set in $\boldsymbol{P}$ to all $(\underline{m}, \underline{n}) \in D_{m,n}$. Under our assumption that each request that enters the first class of (m, n) must return to the downstream after the final class, and assuming stability, the flow from and back to a downstream $(\underline{m}, \underline{n})$ must be equal. We can thus obtain the return routing probability as the fraction of inflow emanating from the downstreams. In the queueing network, the first downstream class $\underline{c}$ to c_{in} belongs to a delay queue, from which the inflow simply becomes $\boldsymbol{x}_{\underline{c}}^T \psi^{\underline{c}}$. Thus, the inflow fraction becomes

$$\boldsymbol{P}_{c_{ret},\underline{c}'}(\boldsymbol{x}) = \frac{\boldsymbol{x}_{\underline{c}}^T \psi^{\underline{c}}}{\sum_{(i,j)\in\mathcal{D}_{m,n}} \boldsymbol{x}_{\underline{d}}^T \psi^{\underline{d}}}, \tag{4.17}$$

where $\underline{c}' = \left[q_d^{\underline{m},m}, (\underline{n}, n), C_2\right]$ when $\underline{c} = \left[q_d^{\underline{m},m}, (\underline{n}, n), C_1\right]$ and $\underline{d} = \left[q_d^{i,m}, (j, n), C_1\right]$. Division by zero can be solved by setting $\boldsymbol{P}_{c_{ret},\underline{c}'}$ to some arbitrary value, since this implies that we currently do not have any flow. The routing matrix $\boldsymbol{P}$ could then be updated with (4.17) for classes where the inflow is affected. If updated, it becomes dependent on the state population.

An alternative solution would be to give each downstream to (m, n) a standalone setup of the corresponding classes in the resulting PS queue. This would simplify the formation of $\boldsymbol{P}^+$, but also lead to a state space explosion in service graphs with longer chains of services with multiple replicas.

Workload Change

The workload changes when the arrival rate of external requests changes. Requests can arrive to our model either via open connections or closed connections. Changes to the open connections are trivial, as $\boldsymbol{\lambda}^+$ can simply be formed with the new rates.

Changes in closed connections can occur via changes in the service time distributions in q_e or in the total number of connecting clients. The distributions are changed by updating the corresponding Ψ^c, ψ^c, ζ^c, and the block diagonal matrices accordingly, to form $\boldsymbol{W}^+$. If predicting transient performance metrics is of interest, it is important that the number of phase states is not changed so that the old values can be used to initialize the new states at t^+. Moreover, the structures of the new phase-type distributions should be similar to the structures of the distributions they are replacing. Otherwise erroneous transients could occur, as the new phase states might then correspond to a highly different internal Markov chain. To add connecting clients, we can simply update $\boldsymbol{x}_c^+(t^+) = \boldsymbol{x}_c(t^+) + \zeta^c Y^c \ \forall c \in \mathcal{C}_{q_e}$ where Y^c is new clients arriving to the client queue class c and ζ^c ensures a correct mapping to the phase states. Clients can in turn be removed by, e.g., direct cancellation of running client requests or departing clients on request completion. Client cancellation can be modeled by updating the affected states as $\boldsymbol{x}_c^+(t^+) = \alpha_c \boldsymbol{x}_c(t^+)$ where $\alpha_c \leq 1$ is some reduction factor to preserve relative internal state flows. Client departures can on the other hand be modeled by removing returning flow to the client waiting queue until the wanted quantity has departed.

Scaling & Migrating

Horizontal scaling changes the number of replicas of any microservice in the service graph, while migration changes the deployment location of a service. For the model, this implies that we may need to change the number of queues, the routing probability matrix $\boldsymbol{P}$ and potentially some service time PH distribution parameters Ψ^c, ψ^c, ζ^c and smoothing parameters $\boldsymbol{\eta}$.

When upscaling, we need to introduce a new PS queue q^+ for the new replica m^+ and new INF queues for the new delays in its connections. Let $m \in \mathcal{M}$ in this section define a replica to the same microservice as m^+. All replicas to a service will have the same sets of upstreams and downstreams, and thus the same number of classes and connecting delay queues.

Service time distributions & smoothing parameter. m^+ will experience some change in request processing speed compared to the other replicas of the same service, but the general characteristics of the service time distributions for the classes should stay similar. Therefore, an easy way to upscale our model if we suspect that m^+ will receive roughly the same service speed as m, is to reuse $(\Psi^c, \psi^c, \zeta^c)$ for all $c \in \mathcal{C}_{q^m}$ in $\mathcal{C}_{q^+}$ and smoothing parame-

ter $\eta^+_{q^+} = \eta_{q^m}$. Otherwise, these will have to be obtained using some other method, e.g., by scaling the PH parameters based on some estimate of the new service speeds.

In the same way, we need to find the PH parameters for all the new delay queues, that is, $(\Psi^c, \psi^c, \zeta^c)\, \forall c \in \bigcup_{\underline{m} \in \mathcal{D}_m} \mathcal{C}_{q_d^{\underline{m},m}} \cup \bigcup_{\overline{m} \in \mathcal{U}_m} \mathcal{C}_{q_d^{m,\overline{m}}}$. If there exists some other m close to m^+ in terms of latency, its delay queues could, for example, be reused for this purpose. Otherwise, if the latencies are measurable, the PH parameters for the delay queues could be obtained from these.

Routing probability matrix. Introducing m^+ also changes $\boldsymbol{P}$, as the downstreams and upstreams for m will now also have to take m^+ into account. We assume that these new routing probabilities at time t^+ are known.

$\boldsymbol{P}$ can then be updated as follows. First, consider the downstreams. For every (m, n) in the replica set of m^+, and for every $(\underline{m}, \underline{n}) \in \mathcal{D}_{m,n}$ where $\underline{u} \in 1 : |\mathcal{C}_{q^{\underline{m}},\underline{n}}| - 1$ is the remote call calling (m, n), let

$$
\begin{aligned}
c_{in} &= \left[q^m, n, C_1\right], & c_{ret} &= \left[q^m, n, C_f\right], \\
\underline{c} &= \left[q^{\underline{m}}, \underline{n}, C_{\underline{u}}\right], & \underline{c}' &= \left[q^{\underline{m}}, \underline{n}, C_{\underline{u}+1}\right], \\
d_1 &= \left[q_d^{\underline{m},m}, (\underline{n}, n), C_1\right], & d_2 &= \left[q_d^{\underline{m},m}, (\underline{n}, n), C_2\right],
\end{aligned}
$$

and set $\boldsymbol{P}^+_{\underline{c},d_1} = \boldsymbol{P}_{\underline{c},d_1}$ which is considered known, $\boldsymbol{P}^+_{d_1,c_{in}} = \boldsymbol{P}^+_{d_2,\underline{c}'} = 1$, and $\boldsymbol{P}^+_{c_{ret},d_2} = \boldsymbol{P}_{c_{ret},d_2}(\boldsymbol{x})$ with (4.17). Similarly, for the upstreams, for all $n \in \mathcal{N}_{m^+}$, $u \in 1 : |\mathcal{C}_{q^+,n}| - 1$ and every upstream $(\overline{m}, \overline{n})$ for u, let

$$
\begin{aligned}
c &= \left[q^+, n, C_u\right], & c' &= \left[q^+, n, C_{u+1}\right], \\
\overline{c}_{in} &= \left[q^{\overline{m}}, \overline{n}, C_1\right], & \overline{c}_{ret} &= \left[q^{\overline{m}}, \overline{n}, C_f\right], \\
d_1 &= \left[d^{m^+,\overline{m}}, (n, \overline{n}), C_1\right], & d_2 &= \left[d^{m^+,\overline{m}}, (n, \overline{n}), C_2\right],
\end{aligned}
$$

and set $\boldsymbol{P}^+_{c,d_1} = \boldsymbol{P}_{c,d_1}$ which is considered known, $\boldsymbol{P}^+_{d_1,\overline{c}_{in}} = \boldsymbol{P}^+_{d_2,c'} = 1$, and $\boldsymbol{P}^+_{\overline{c}_{ret},d_2} = \boldsymbol{P}_{\overline{c}_{ret},d_2}(\boldsymbol{x})$ with (4.17).

Final parameters. As we introduce a set of new empty queues, we get $\boldsymbol{x}^+_c(t^+) = \boldsymbol{x}_c(t^+)$ for all old classes and $\boldsymbol{x}^+_c(t^+) = 0$ for all newly introduced classes. From our new set of classes and their PH distributions, we can extend the block diagonal matrices $\boldsymbol{\Psi}, \boldsymbol{A}, \boldsymbol{B}$, and together with $\boldsymbol{P}^+(\boldsymbol{x})$ form $\boldsymbol{W}^+(\boldsymbol{x})$. The place where we introduce these new blocks is not important, as long as the order of classes is the same in all three and $\boldsymbol{P}^+$, $\boldsymbol{x}^+(t^+)$, and $\boldsymbol{\eta}^+$.

Migration, downscaling, & vertical scaling. Migration can be implemented as a special case of upscaling by simply copying the routing probabilities associated with the old to-be-migrated replica to the newly created replica, and subsequently setting the routing probabilities to 0 from all downstreams to the old replica. After some time, the class populations in the queue

corresponding to this old replica will be empty, and the corresponding states can be removed. Similarly, downscaling can be implemented by directly setting the routing probabilities to 0 from all downstreams to the to-be-removed replica. After some time, class populations in the corresponding queue will be empty, and the states can be removed.

Finally, vertical scaling is difficult to implement without assuming more knowledge of the system. We need to know how scaling a resource to a service affects the service time distributions in each class to perform any nontrivial predictions. We will leave this for future work.

4.3 A Federated Application Sandbox

In this section, the federated application (FedApp) sandbox is introduced, on top of which the microservice fluid model is later evaluated. The sandbox was created to provide an experimental multi-cloud environment that (i) is flexible, yet remains a close approximation of real systems, (ii) is easy to both deploy and use, and (iii) has an easily extendable implementation. Its key features include:

- Creation of a user defined, multi-cluster virtual environment in an OpenStack cloud, with a centralized structure complete with tools for controlling and monitoring the entire federation, streamlining the setup and usage of the sandbox.
- Possibility of inducing network characteristics such as delay, jitter, and packet loss between clusters, thus enabling faithful emulation of real world environments.
- Application orchestration using Kubernetes complete with a multi-cluster service mesh layer using Istio, for easy handling and deployment of applications spanning multiple clusters.

Experimentally evaluating performance models and management methods for cloud applications in real environments is an important, yet nontrivial endeavor. Constructing the necessary environment is, in general, difficult and time consuming, and quickly becomes complicated when dealing with modern trends such as the microservice architecture or multi-cloud deployments. Although different simulation or emulation tools [Ahmed and Sabyasachi, 2014; Abreu et al., 2019; Casale, 2020] can play a crucial role in initial design and evaluation of such models and methods, they ultimately fail to capture the actual behavior of the system.

The sandbox is released as open source and is available at GitHub[3].

[3] https://github.com/JohanRuuskanen/FedApp

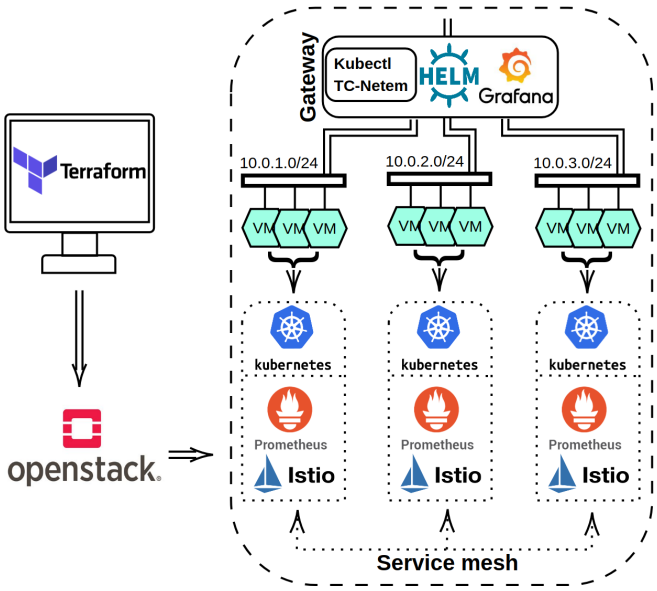


Figure 4.4 Illustration of an example setup of the complete sandbox with 3 clusters containing 3 virtual machines each.

Sandbox Design

The design choices made for creating the environment to fulfill the stated goals are presented here. An illustration of the complete setup can be seen in Figure 4.4 which can be used as an overview of the sandbox in its entirety.

Topology of the multi-cluster environment. The clusters constitute the foundation of the sandbox, and they will need some kind of infrastructure to be realized upon. To provide this in a both flexible and easy to use manner, we chose to supply the means of creating a suitable virtual infrastructure in OpenStack, an open source cloud platform commonly used for providing IaaS in private clouds. This provides both a controlled environment in which the rest of the sandbox can be deployed in a standardized manner, and makes it flexible as it is possible to easily scale the size and amount of clusters.

To emulate a multi-cluster environment, each cluster is first provisioned as a set of virtual machines with their own isolated network. Cluster-to-cluster communication is then enabled by connecting each internal cluster network to a *gateway* virtual machine. At first, this might seem odd as we are in fact introducing a potential bottleneck. But given the stated goal of the sandbox, centralizing the network through a VM in this manner is beneficial. First, from a single point it enables network characteristics to be

affected and network behavior to be observed. Moreover, a centralized VM increases usability as it can be deployed in a standardized manner hosting all the tools necessary for controlling, monitoring, and deploying applications on the clusters. Finally, the choice of pooling all functionalities for managing the sandbox makes the setup easy to extend and easy to debug when something eventually breaks down.

The sandbox comes supplied with scripts to first deploy the gateway virtual machine using Terraform[4] and then to provide it with all the necessary tools using Ansible[5]. The rest of the sandbox, including the virtual clusters, is then deployed from the gateway VM.

Network characteristics emulation. To provide a close approximation to a real multi-cloud environment, it is vital that desired network characteristics between clusters can be imposed. Benefiting from the chosen centralized topology, the procedure for mimicking a real world networking environment is greatly simplified. Each virtual cluster sees itself as a "standalone" cluster on a private network, where all the intercluster communication traffic is handled by the gateway VM. Thus, by only affecting the intercluster routing logic on the gateway, an arbitrary network profile can be emulated.

We achieve this emulation by using the Linux Traffic Control (TC) utility on the gateway VM. The TC network emulator (TC-netem)[6] provides the possibility of adding packet loss, delay and other characteristics on the packets from a selected Network Interface Controller (NIC). By applying desired emulated characteristics on the NICs of the Gateway VM, the network characteristic to a cluster can be arbitrarily specified, for example, to mimic the propagation and transmission delay caused by the geographic distance.

Furthermore, by utilizing TC qdisc and filter, it is possible to control the point-to-point network characteristics between clusters. Each cluster-to-cluster connection can be given its own network characteristic configuration, without affecting neither connections between other clusters nor important meta-communication between the Gateway VM and the clusters, such as monitoring data or control commands.

The sandbox comes supplied with scripts using TC-netem to make this cluster-to-cluster network characteristic emulation straightforward to use.

Deploying & managing multi-cloud applications. The services in a microservice application are often deployed in standalone containers, which communicate using some communication protocol such as http. To allow these applications to be efficiently managed in a realistic manner, the sandbox fits each cluster with the well-known container orchestrator Kubernetes[7].

[4] https://github.com/hashicorp/terraform
[5] https://github.com/ansible/ansible
[6] https://www.linux.org/docs/man8/tc-netem.html
[7] https://kubernetes.io/

This creates an abstraction layer between the raw virtual infrastructure and the application, providing a platform that simplifies the deployment and management of complex applications. Furthermore, as Kubernetes pools its resources and allows multiple users to quickly deploy/destroy containers in this pool, each virtualized cluster can essentially be viewed as a small private cloud in some sense, providing a PaaS. To enable nontrivial collaboration in applications spanning multiple of these "clouds", the sandbox also deploys Istio[8] on top of the Kubernetes instances, which greatly simplifies intercluster service-to-service networking.

To provide insight on the behavior of the multi-cloud environment and its applications, the sandbox provides federation-wide monitoring in the form of data collection using Prometheus[9], and a Grafana[10] GUI deployed on the Gateway VM. With this setup, it is possible to inspect important metrics, such as CPU and memory utilization, out-of-the-box in real time for all parts of the deployed sandbox from the Gateway VM. Furthermore, the Istio service mesh makes it possible to retrieve the access logs required for distributed tracing for every request over all service replicas.

In the sandbox, scripts are supplied that utilize the tool Kubespray[11] to deploy Kubernetes on each virtual cluster. Further, via these scripts, Istio can be deployed preconfigured to support the multi-cluster service mesh, and the Prometheus/Grafana stack across the entire multi-cluster environment.

Functionalities

Here follows a quick summary on the functionalities the sandbox design provides, in order to fulfil the goal of providing an an easy-to-use multi-cloud environment for application experimentation.

First and foremost, the user defined virtual clusters together with the cluster-to-cluster network emulation can be used to emulate a wide range of different multi-cloud or fog/edge computing settings. Further, Kubernetes and Istio simplifies the effort to deploy and manage multi-cloud applications in these settings. Also, with Istio and Prometheus, request tracing and monitoring of resource usage becomes easy. Moreover, the usage of well-known cluster software, alongside pooling the tools to manage them in the central gateway VM, makes it easy to incorporate new tools or software that are not supplied out-of-the-box. Finally, given access to an OpenStack cloud, the effort of deploying the entire sandbox becomes simple and flexible thanks to the standardized environment that the virtualized infrastructure can provide.

The possible use cases of the sandbox are numerous, and many of the open issues in cloud computing as discussed in [Buyya et al., 2018] could essentially

[8] https://istio.io/
[9] https://github.com/prometheus/prometheus
[10] https://github.com/grafana/grafana
[11] https://github.com/kubernetes-sigs/kubespray

be explored in it. For example, different methods for reliability, resource management and scheduling of microservices distributed over multiple clouds can be readily investigated.

Limitations. The sandbox is, however, not without its limitations. First, it assumes access to an OpenStack cloud, which is not something that is freely accessible for everyone. Other IaaS providers could be used for the virtual infrastructure, but this would require adaptation of the Terraform scripts.

Furthermore, the centralized network topology, although good for our goal, will be problematic if we try experiments of too many clusters or a too high intercluster load. In these cases, cluster-to-cluster network characteristics could potentially be implemented distributively, by connecting the cluster networks directly and using TC-netem directly on each cluster.

4.4 Experimental Evaluation

To test the validity of our model, we developed a simple example application that performs face detection as a service based on three microservices. The application was then deployed in the FedApp sandbox and loaded with a stream of images, to test how well the fluid model was able to model and predict its performance metrics.

Example Application

The three microservices that constitute the application are (i) `frontend` for image pre-/postprocessing, (ii) `backend` for actual face detection, and (iii) `storage` for storing results. `frontend` has two http endpoints: `detect/` for detecting a face in an image and `fetch/` for fetching an already detected face from the storage. Furthermore, `backend` has a single http endpoint `detect/` for detecting a face in a preprocessed image. Finally, `storage` has two http endpoints, `store/` for storing a preprocessed image and `fetch/` for retrieving a stored image in an internal Redis[12] database. Summarized, the microservices have the following http endpoints, which in turn invoke the following remote calls:

$$
\begin{aligned}
\texttt{frontend:}\quad & \begin{cases} \texttt{detect/} & \rightarrow \texttt{backend/detect/} \\ \texttt{fetch/} & \rightarrow \texttt{storage/fetch/} \end{cases} \\
\texttt{backend:}\quad & \begin{cases} \texttt{detect/} & \rightarrow \texttt{storage/store/} \end{cases} \\
\texttt{storage:}\quad & \begin{cases} \texttt{fetch/} \\ \texttt{store/} \end{cases}
\end{aligned}
$$

[12] https://github.com/redis/redis

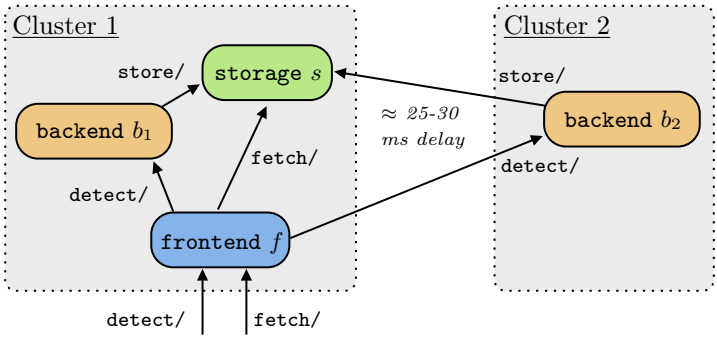


Figure 4.5 The experimental setup of the example application with three microservices. The `backend` service has one replica in each of the two clusters. Its queueing network model translation is shown in Figure 4.2.

We chose to design our own simple application instead of relying on existing benchmarks, such as the well-known *SockShop*[13] or *DeathStarBench*[14], for multiple reasons: (i) It is easier to reason about the results from a smaller example system, (ii) deployment and validation of correctness are simplified in our limited experimental setup, and (iii) it took less effort to ensure that the system is robust to longer periods of heavy traffic. Despite its apparent simplicity, the application captures a system with multiple endpoints, a network intensive service (`frontend`), a CPU intensive service (`backend`), and a memory intensive service (`storage`).

Experimental Setup

All three microservices are implemented in Python using Flask[15] and deployed using Gunicorn[16] to enable robust and easy multi-threaded service of incoming requests. Every request is served in a standalone thread that is blocked while calling upstream services. To avoid bottlenecks, the services were tuned to yield roughly similar utilization for a range of input loads.

To mimic the deployment in a real software stack and test the model in a more interesting setting, the application was deployed on the FedApp sandbox introduced in Section 4.3, using two virtual clusters. Each cluster was given 4 virtual machines, each with 4 vCPU and 8 Gb of RAM. Furthermore, an additional 25-ms delay was added to all traffic between the two clusters to emulate a geographical distance. In the first cluster, one replica of each of the three microservices was deployed. In the second cluster, a second replica

[13] https://github.com/microservices-demo/microservices-demo
[14] https://github.com/delimitrou/DeathStarBench
[15] https://github.com/pallets/flask
[16] https://github.com/benoitc/gunicorn

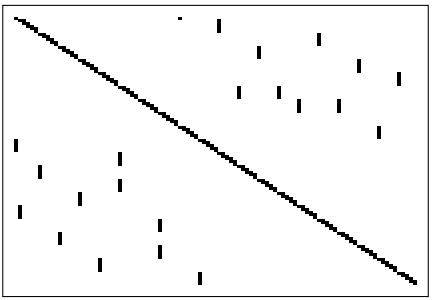


Figure 4.6 Sparsity pattern of the $\boldsymbol{W} \in \mathbb{R}^{101\times101}$ matrix from the example application. The black squares symbolizes nonzero elements in the matrix.

of `backend` was deployed to perform remote offloading. A random load balancer was used to balance requests between the two backends. The resulting application deployment is illustrated in Figure 4.5, and as can be seen its service graph becomes the same as the illustration in Figure 4.2.

The resulting model has $|\mathcal{Q}| = 10$ queues, of which four are PS queues with $k_m = 4$, five are delay queues, and one is the client queue. The PS queues have four classes for the `frontend`, and two classes for the `backend` and `storage`, resulting in a total of $|\mathcal{C}| = 21$ classes. For all of our experiments presented, each class $c \in \mathcal{C}$ was given $|\mathcal{S}_c| = 5$ phase states, while the client queue class was given 1 phase state, for a total of $|\mathcal{S}| = 101$. This is a fairly large number of states, but the resulting $\boldsymbol{W}$ becomes sparse as we fit the PH distributions to the service times using a Coxian structure [Cox, 1955] with the EM algorithm from `EMpht.jl`[17]. The sparsity of the resulting model is shown in Figure 4.6.

The code for the experiments is available on GitHub[18].

Results

We wanted to test both how well the smoothed mean-field fluid model was able to capture different performance metrics and how well it was able to predict performance metrics after a system perturbation. Therefore, we conducted two different types of experiments. In the first, the effect of perturbing the workload was studied. In the second, the effect of horizontal scaling and migration was studied. For all experiments, we compared the results of the data with two smoothed mean-field fluid models; one fitted in the current operating condition (denoted $\mathcal{F}_{fit}$) and one fitted in some other operating condition for predictive purposes (denoted $\mathcal{F}_{pred}$). The measured stationary

[17] https://github.com/Pat-Laub/EMpht.jl
[18] https://github.com/JohanRuuskanen/Cloud2022_code

mean queue length X_q^* and the 95th percentile of response times φ were then compared with their equivalents generated from $\mathcal{F}_{fit}$ and $\mathcal{F}_{pred}$. Transient values of the mean queue lengths were not considered as they are difficult and time consuming to evaluate in a real experimental setting.

Experiments were carried out using a load generator to strain the system simultaneously with an open connection to the `frontend/detect/` endpoint with a rate λ and a closed connection to the `frontend/fetch/` endpoint with 50 clients each with an exponentially distributed waiting time with mean $1/\mu_c$. The images used for the requests were sampled from the UMass face detection data set and benchmark [Jain and Learned-Miller, 2010]. After each experiment, the tracing data were extracted from Istio and used to fit the queueing network model and the smoothing parameter for the fluid model. For the predictions, we transformed the models according to Section 4.2, but used the estimates of λ, μ_c, and P from the data if not noted otherwise. Although these estimates are similar to the desired values, they vary slightly due to the dynamics of the cloud and the implementation of our experiment application.

Workload perturbations. Here, we carried out 16 different experiments, each for 300 seconds on a grid of workload rates $\boldsymbol{\lambda} = (8, 16, 24, 32)$ and $\boldsymbol{\mu_c} = (0.2, 0.8, 1.4, 2)$. In Table 4.2 the estimated utilization from the data for each grid point can be seen, which shows that the experiments capture utilizations ranging from low to high. $\mathcal{F}_{pred}$ was fitted to the data from Experiment 11. As can be seen, increasing λ increases the utilization in all services as the `detect/` endpoint activates all services, while increasing μ_c only increases utilization in `frontend` and `storage` as these are the only ones activated by the `fetch/` endpoint.

The mean request populations and 95th response time percentiles over

Table 4.2 Estimated utilization for each service for each of the 16 experiments when perturbing the workload.

$\hat{\rho}_{b1}$, $\hat{\rho}_{b2}$ $\hat{\rho}_f$, $\hat{\rho}_s$	$\boldsymbol{\lambda^1}$	$\boldsymbol{\lambda^2}$	$\boldsymbol{\lambda^3}$	$\boldsymbol{\lambda^4}$
$\boldsymbol{\mu_c^1}$	.16, .20, .18, .22	.42, .37, .24, .27	.63, .64, .30, .30	.85, .87, .34, .34
$\boldsymbol{\mu_c^2}$	.16, .17, .40, .60	.36, .39, .51, .61	.56, .62, .54, .61	.88, .86, .62, .64
$\boldsymbol{\mu_c^3}$	.16, .17, .65, .78	.38, .34, .70, .80	.63, .58, .74, .79	.84, .83, .78, .82
$\boldsymbol{\mu_c^4}$	.15, .16, .76, .87	.34, .39, .80, .88	.53, .63, .83, .89	.81, .86, .89, .90

the two endpoints for the two fluid models are compared with data in Figure 4.7 for absolute values over the entire application and in Figure 4.8 for absolute errors in the services. In general, $\mathcal{F}_{fit}$ manages to approximate these performance metrics very well, except for the 95th percentiles of the `frontend/detect/` endpoint at the highest utilization levels, as seen in Figure 4.7(c). This is to be expected, since the response time CDF approximation relies on the assumption that every request receives the mean processor share. For queues under Poisson arrivals, this assumption becomes less accurate the higher the utilization becomes. A high utlization implies a high queue length variance and in turn a higher variance for the processor shares.

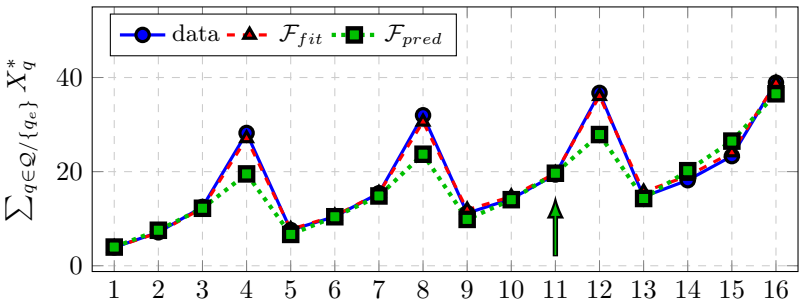
In the same way, $\mathcal{F}_{pred}$ manages to capture the performance metrics fairly well in all perturbations tested, except at higher utilizations. For such experiments, $\mathcal{F}_{pred}$ provides a less accurate mean request population, as can be seen in Figure 4.7(a) and Figure 4.8(a), where errors are concentrated to the two backends and the storage. Due to this, the 95th percentiles are also prone to higher errors at high utilizations, which are more pronounced for the `frontend/detect/` endpoint.

Horizontal scaling and migration. Here, we carried out 11 different experiments with rates $\lambda = 13$ and $\mu_c = 1.4$. The weights of the random load balancer was changed in incremental steps, from sending all requests to `backend 1` in Experiment 1 to sending all to `backend 2` in Experiment 11. The probabilities and estimated utilizations for each experiment can be seen in Table 4.3. As can be seen, the chosen values yields a high utilization when all requests are directed to a single backend. Furthermore, more requests to `backend 2` also increase the utilization of `frontend`, probably because it needs to handle the extra delay.

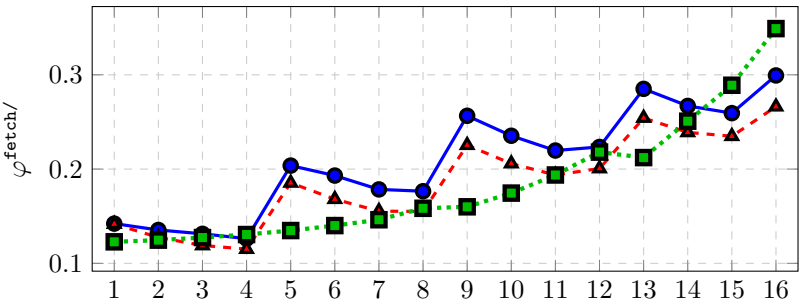
By perturbing the load balancing like this, we can test how well our model can predict the effects of both service migration and horizontal scaling. For migration and upscaling, we introduce $\mathcal{F}^1_{pred}$, which we fit in Experiment 1 where all requests are sent to `backend 1`. Since we do not have data on

Table 4.3 Routing probabilities and estimated utilization for each service for each of the 11 experiments when perturbing the load balancing.

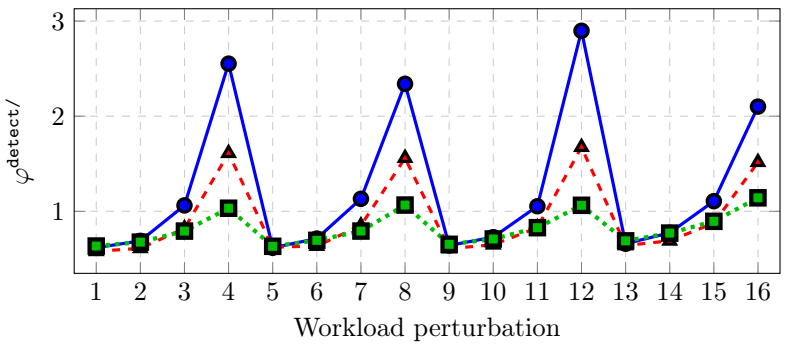
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$\boldsymbol{P}_{f,b1}$	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1	0.0
$\hat{\boldsymbol{\rho}}_{\boldsymbol{b1}}$	.86	.75	.63	.50	.42	.31	.25	.16	.10	.04	0.0
$\hat{\boldsymbol{\rho}}_{\boldsymbol{b2}}$	0.0	.05	.11	.20	.27	.34	.42	.52	.66	.71	.83
$\hat{\boldsymbol{\rho}}_{\boldsymbol{f}}$	.55	.60	.63	.65	.67	.69	.70	.72	.73	.73	.73
$\hat{\boldsymbol{\rho}}_{\boldsymbol{s}}$	.74	.77	.79	.79	.79	.79	.77	.77	.76	.75	.76



(a) Mean request population in the entire application.

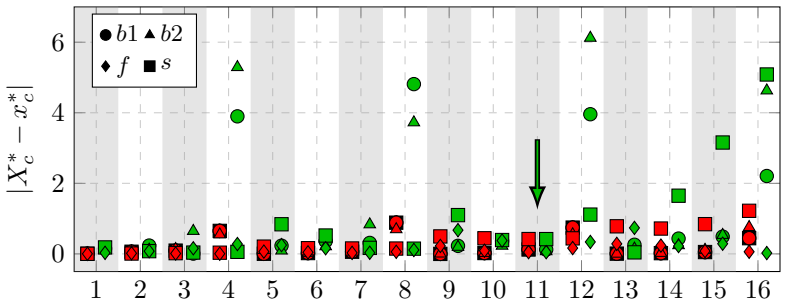


(b) Response time p95 in seconds concerning the `fetch/` endpoint.

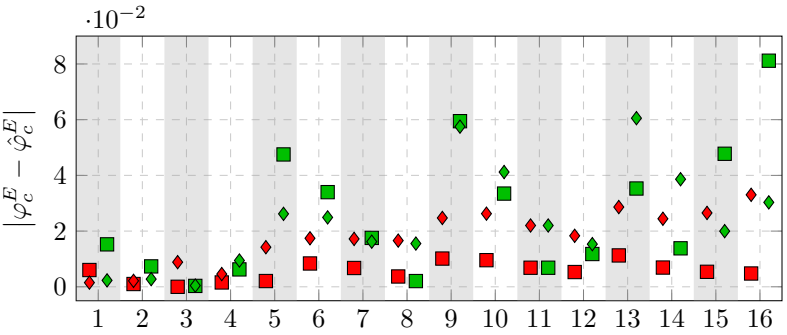


(c) Response time p95 in seconds concerning the `detect/` endpoint.

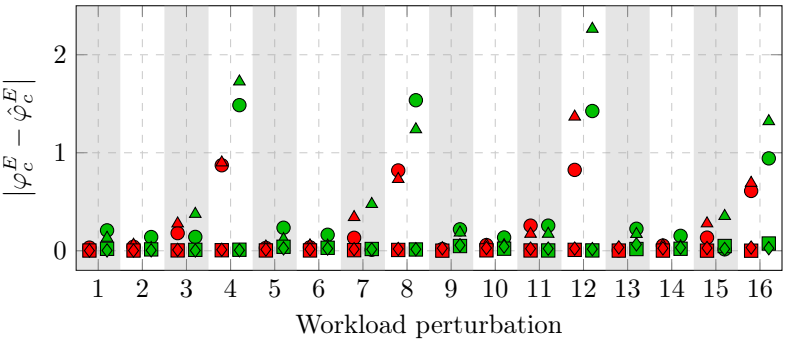
Figure 4.7 Results from the 16 workload perturbation experiments considering gathered data (blue line), $\mathcal{F}_{fit}$ (red dashed) and $\mathcal{F}_{pred}$ (green dotted). The arrow indicates at what experiment $\mathcal{F}_{pred}$ has been fitted. In (a) the total mean requests over the entire example application is shown. In (b) and (c) the 95th percentile of the response times for requests visiting the `fetch/` and `detect/` endpoints are shown.



(a) Total population absolute error in each service.

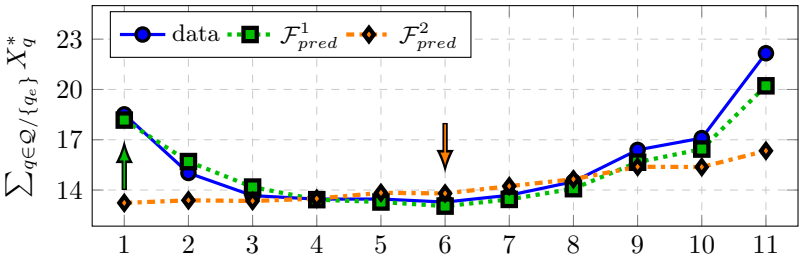


(b) Response time p95 absolute error concerning the `fetch/` endpoint.

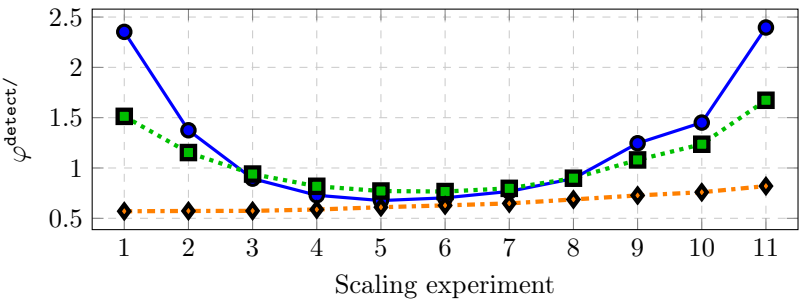


(c) Response time p95 absolute error concerning the `detect/` endpoint.

Figure 4.8 Results from the 16 workload perturbation experiments considering $\mathcal{F}_{fit}$ (red marks) and $\mathcal{F}_{pred}$ (green marks). The arrow indicates at what experiment $\mathcal{F}_{pred}$ has been fitted. In (a) the absolute error over the mean requests in the entire example application is shown. In (b) and (c) the absolute error over the 95th percentile of the response times for requests visiting the `fetch/` and `detect/` endpoints are shown. Here, each marker type represents a service as noted in the legend.



(a) Mean request population in the entire application.

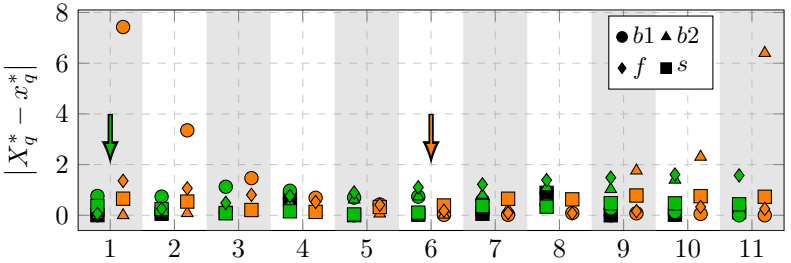


(b) Response time p95 in seconds concerning the `detect/` endpoint.

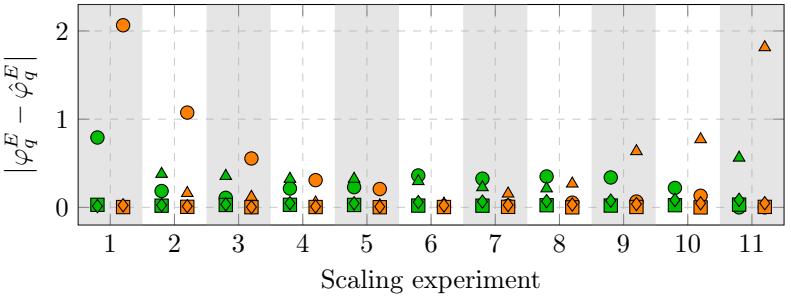
Figure 4.9 Results from the 11 scaling experiments considering gathered data (blue line), $\mathcal{F}^1_{pred}$ (green dotted), $\mathcal{F}^2_{pred}$ (orange dash-dotted). The arrows indicate at what experiments $\mathcal{F}^1_{pred}$ and $\mathcal{F}^2_{pred}$ has been fitted. In (a) the total mean requests over the entire example application is shown. In (c) the 95th percentile of the response times for requests visiting the `detect/` endpoint is shown.

`backend 2` at this point, we reused the PH distributions and the smoothing value of `backend 1` for the fluid model. Furthermore, we set the service time distributions in the delay queues to and from `backend 2` as Erlang distributions that match the mean RTT / 2. Finally, the return routing probabilities from `storage/store/` were updated according to (4.17). For downscaling, we instead introduce $\mathcal{F}^2_{pred}$, which we fit in Experiment 6 where the load balancing weights are equal.

The result of the experiments are shown in Figure 4.9 for the absolute values over the entire application and in Figure 4.10 for the absolute error in the services. Here, we only consider the mean request populations and the 95th percentiles for the `detect/` endpoint. As can be seen, $\mathcal{F}^1_{pred}$ is quite accurate and also coincides largely with $\mathcal{F}_{fit}$ (not shown). Transferring the



(a) Total population absolute error in each service.



(b) Response time p95 absolute error concerning the `detect/` endpoint.

Figure 4.10 Results from the 11 scaling experiments considering $\mathcal{F}^1_{pred}$ (green marks) and $\mathcal{F}^2_{pred}$ (orange marks). The arrows indicate at what experiments $\mathcal{F}^1_{pred}$ and $\mathcal{F}^2_{pred}$ has been fitted. In (a) the absolute error over the total mean requests in the entire example application is shown. In (c) the absolute error 95th percentile of the response times for requests visiting the `detect/` endpoint is shown. Here, each marker type represents a service as noted in the legend.

PH distributions and the smoothing parameter from one replica to another seems to work very well when predicting these performance metrics. However, the results would be worse if there were a significant change in service speed at the new replica location. Finally, it can be seen that $\mathcal{F}^2_{pred}$ is unable to accurately predict the downscaling to `backend 1` in Experiment 1 or `backend 2` in Experiment 11. For both prediction models, the errors seem to be concentrated in the two backends.

4.5 Summary and Discussion

In this chapter, we have presented a simple yet general mixed queueing network model for microservice applications, which can be quickly solved using the smoothed mean-field fluid model. Each service is modeled as a multi-class processor sharing queue, each service-to-service delay is modeled as a delay queue, and each service time is allowed to have a phase-type distribution. We further showed that the model parameters can be extracted online in a distributed fashion from common tracing data.

Furthermore, we introduced the FedApp sandbox which simplifies the effort to setup of a multi-cluster environment in OpenStack with the possibility of introducing arbitrary network characteristics between clusters. This sandbox was then used to perform experiments on an example application, which showed that the model can be used to quickly obtain accurate estimates and predictions of important performance metrics in certain cases.

Discussion on the Experimental Results

From the two sets of experiments, it is evident that there exist some unmodeled load dependencies in the processing speeds and interactions between the two request types. This is to be expected, as modeling each service as a single multi-class PS queue is quite coarse. Despite this, at least when services are implemented to handle requests in a PS-like manner, the model can generate accurate estimations of different performance metrics for the current operating condition. This implies that we can use the model to accurately predict performance metrics for small system perturbations.

For larger perturbations, we noticed that the predictions over the workload experiments were accurate as long as we predict in directions that do not result in any major increase in utilization. This is mirrored in the scaling and migration experiments. Our upscaling and migration only give a moderate increase in the `frontend` utilization from the fitted operating condition. On the contrary, our downscaling significantly increases utilization in either of the two `backend` replicas. As can be seen, the prediction in upscaling and migration performs well, while for downscaling, it performs poorly. This suggests that general predictions for large perturbations could be accurate if we were able to fit the model at high utilization.

Since the prediction errors in both experiment types seem to be mostly concentrated on the two `backend` replicas, which are subject to requests from an open connection, a probable cause for this is the fitting of our smoothing parameters. This parameter aims to capture the queue population variability, and for classes under open connections, this variability increases sharply at high utilizations. Therefore, since we fit the smoothing parameter according to (3.35), we will find a good smoothing value for the current operating condition that will no longer be valid if the utilization increases too much.

However, it seems that the value found at high utilization can produce adequate predictions for lower utilizations. Furthermore, due to model errors, the service time distributions were found to be load dependent, but only slightly. This is why reusing them for prediction seems to generate only smaller errors. However, if this load dependency were larger, such as if the system had services with very non-PS-like handling of the request, then predicting in a direction of decreasing utilization would most probably be problematic as well.

Model Limitations and Errors

The introduced model has some limitations in terms of the microservice graphs it can capture and potential error sources if the necessary assumptions are not fulfilled.

Limitations. The model requires some assumptions on the connection between request types in services, i.e., every request in (m, n) must have the same number of remote calls and multiple downstreams to (m, n) are only allowed if the downstreams are replicas of the same microservice. However, since "request type" is a user defined classification of requests visiting a service, these assumptions can still be fulfilled by simply introducing additional request types when violated.

However, the model has some hard limitations. Remote calls must be sequential, and each remote call is assumed to be synchronous implying that the processing of the calling request is paused until a response is returned. In addition, each request can call only a single upstream in each remote call. In particular, this disallows modeling of systems with fork-join behavior.

Still, simple forking of requests could probably be supported in the fluid model, as long as the returning responses can be correctly handled. This would allow for the modeling of multiple simultaneous calls, *asynchronous* calls and *second-phase processing* in certain cases. An asynchronous call implies that request processing is not blocked after the call is issued. This could be modeled by forking the request issuing the call into one outgoing request bound for an upstream class/queue, and one request that directly joins the next class in the queue. Moreover, for services with second-phase processing, where the processing of a request continues after a response has been returned, this second phase could be modeled as a fork from the returning response that continues being served in the queue. After the forks modeling the asynchronous call or second-phase processing completes their service, they would have to depart the queueing network.

Error sources. Certain assumptions are not breaking but rather introduce modeling errors when violated. First and foremost, as mentioned above, the model assumes that each service can be adequately modeled as a single PS queue, which can be more or less true depending on how the application

is implemented. Furthermore, the inability to capture request processing in a service during a remote call, or after a response has been returned, will create errors if these are present. In addition, the model assumes that every load balancer between multiple upstreams uses the weighted random policy. If another policy is used, modeling errors will be introduced. Finally, the model assumes that each service from an open connection arrives according to a pure Poisson process. However, this could potentially be remedied by *closing* the open connection [Bolch et al., 2006, p. 507].

5

Holistic Load Balancing via Fluid Model Differentiation

Runtime management of cloud applications is necessary to balance cost minimization while adhering to SLO constraints. Modern trends such as the microservice architecture and deployments in fog/edge computing environments add new difficulties to this balancing act. Applications can consist of many small services interacting in a graph structure, where each service can be scaled individually and potentially placed on different sites. Requests accessing the application will traverse this graph over different paths, depending on both the type of request and the load balancing strategy chosen between replicas. Due to differences in, e.g., communication delays, available capacity, and resource costs, the choice of request paths will influence the overall application latency and cost. These can, in turn, be controlled by tuning the load balancers located between each communicating replica set.

Introduction

In this chapter, we demonstrate how automatic differentiation over the microservice fluid model introduced in Chapter 4 can be used to optimize a running application, by tuning the load balancers to minimize some user defined, *holistic* cost function under response time percentile constraints. We assume that these load balancers use a weighted random strategy, and hence the control action boils down to choosing the routing probabilities between replicas. Furthermore, a multi-cloud deployment as seen in Figure 5.1 is assumed, where replicas of the same service can be placed on different sites.

In summary, the method works as follows. First, a cost function and constraints are introduced based on performance metrics retrievable from the fluid model. Then, the microservice fluid model of Chapter 4 is extracted by parsing the tracing data of the application. By using automatic differentiation, the cost function gradient can be obtained with respect to the load balancing probabilities. This enables us to update these parameters, using,

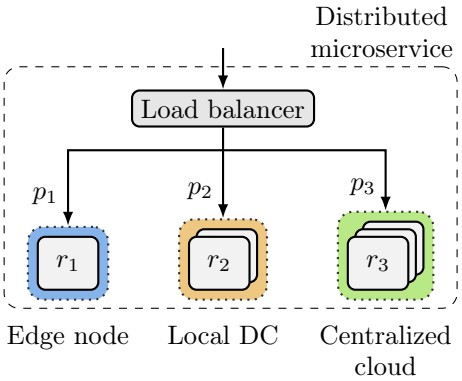


Figure 5.1 A microservice with its replica sets r_i over multiple different sites. The load balancer distributes the incoming load according to a weighted random scheme with probabilities p_i over the available replicas. Multiple such microservices can be joined to deliver a larger application.

e.g., gradient stepping, in a manner that steers the application towards a setting of less cost while adhering to the constraints. As the model extraction and gradient calculation are relatively cheap, new parameters can be decided iteratively on a running application to ensure that the system can react to disturbances and is updated in a robust manner.

In an experimental evaluation on the small microservice application introduced in Section 4.3, it is shown that the method can quickly step towards optimal values while supporting complicated cost functions based on the solutions to a system of ODEs.

Outline. First, the microservice fluid model is quickly recapped and the concept of automatic differentiation is introduced. In Section 5.1, we discuss how to extract the desired performance metrics and relate them to the load balancing parameters. Then, in Section 5.2, these performance metrics are used to define the cost function to be differentiated, and the cost optimizing algorithm for the load balancing parameters is derived. In Section 5.3, the method is experimentally evaluated on the small multi-cloud microservice application. Finally, the chapter is summarized in Section 5.4.

Assumptions and Notations

The microservice fluid model introduced in Chapter 4 constructs a simplistic queueing network model of the application that can be quickly solved using the smoothed mean-field fluid model from Definition 3.2. The model represents each microservice replica as a single multi-class PS queue and each replica-to-replica communication delay as a multi-class INF queue. This al-

lows the processing stages visited by a request across the entire microservice application to be modeled as a path over the classes in the network. Each class is assumed to have a phase-type distributed service time. Furthermore, external requests are assumed to arrive either as Poisson arrivals or as clients utilizing the application. Although the individual replica models are simple, this fluid model can capture quite general graphs of microservices, and be completely extracted at runtime from commonly collected tracing data denoted $\mathcal{H}$. However, due to its simplicity, it can experience modeling errors when predicting too far from the operating point where it was fitted.

A similar notation to Chapter 3 and Chapter 4 will be adopted in this chapter. Let $\mathcal{Q}$ be the set of queues, $\mathcal{C}$ the set of classes, and $\mathcal{S}$ the set of phase states in the network. Each queue q is assumed to have a unique set of classes $\mathcal{C}_q$, and each queue/class pair $(q, c) \in (\mathcal{Q}, \mathcal{C}_q)$ is assumed to have a unique set of phase states $\mathcal{S}_{q,c}$. Furthermore, let k_q be the number of servers in queue q, $\boldsymbol{\lambda} \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ the Poisson arrival rates to each class and $\boldsymbol{P} \in \mathbb{R}^{|\mathcal{C}|\times|\mathcal{C}|}$ the class-to-class routing probability matrix. Finally, let $\boldsymbol{\Psi} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}, \boldsymbol{B} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}, \boldsymbol{A} \in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}$ be the parameter matrices for the PH distributions of each class, stacked into block diagonals in the appropriate order. Let $\boldsymbol{X}(t) \in \mathbb{R}^{|\mathcal{S}|\times 1}$ be the population vector of all phase states at time t. The smoothed mean-field fluid model approximates $\mathbb{E}[\boldsymbol{X}(t)]$ with the solution $\boldsymbol{x}(t) \in \mathbb{R}^{|\mathcal{S}|\times 1}$ to the following system of ODEs,

$$\begin{aligned} \frac{d\boldsymbol{x}}{dt} &= \boldsymbol{W}^T \hat{\theta}(\boldsymbol{x}, \boldsymbol{\eta}) + \boldsymbol{A}\boldsymbol{\lambda}, \\ \boldsymbol{x}(0) &= \boldsymbol{X}(0), \quad \boldsymbol{\eta}(t) > \boldsymbol{0}, \end{aligned} \tag{5.1}$$

where $\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T$, $\hat{\theta}_i(\boldsymbol{x}, \boldsymbol{\eta}) = x_i \hat{g}_{Q(i)}\left(\boldsymbol{x}, \eta_{Q(i)}\right)$, and $\hat{g}_q(\boldsymbol{x}, \eta_q)$ is defined as (3.31) for some phase state i and queue q.

Automatic Differentiation

Automatic differentiation is a technique for evaluating the derivatives of functions defined by computer programs. The basic idea is to apply the chain rule to the code in order to reduce it to simpler expressions where the derivative of each individual operation can be easily defined. There are a few different methods to automate this process, for this work we chose an existing implementation based on *dual numbers* [Eastham, 1961].

Dual numbers conveniently allow for propagation of information regarding the value of an expression, as well as the derivative of the expression, at the same time. Hence, if we have a program where the higher-level code is agnostic to type, and the lower level operations are defined for dual numbers, both the value and the derivative can be calculated in one go by supplying parameters as dual numbers.

Automatic differentiation distinguishes itself from numerical differentiation by being *exact* in the mathematical sense. While numerical differentiation approximates derivatives using finite differences, and thus suffers from the inaccuracies coming from those techniques, automatic differentiation utilizes the exact expression. Furthermore, numerical differentiation has the disadvantage of being computationally inefficient as the number of variables grows, especially for higher-order derivatives [Rall, 1981].

5.1 Microservice Application Model

We consider a cloud application subjected to requests from external users and where there are (soft) constraints on certain performance metrics, e.g., response time percentiles, via SLOs. It is assumed that the application consists of multiple microservices that interact in a graph structure to deliver the full application. Each microservice is assumed to have a set of replicas that are allowed to span multiple placement possibilities, e.g., machines, clusters, or even different sites, each associated with their own communication delay. The service of requests incurs a certain *cost* for the application owner, depending on things such as the cost of electricity, availability, the specific cloud provider, etc. This cost is highly specific to the application and the deployment, but we will assume that it is related to *where* the requests are executed among the placement possibilities.

The end goal is to minimize the total cost of running the application, while not violating the SLO constraints, by tuning parameters related to the application deployment and management. To effectively determine these parameters, a model can be used to estimate the impact of them on both cost and constraints. Moreover, given such a model, it is possible to use automatic differentiation to differentiate the cost derived through this model with respect to these parameters. Potentially, one could then devise a control strategy to steer the entire application towards an operating region of less cost while keeping clear of the constraint limits. This procedure is exemplified in this chapter by considering the load balancing between the different replica sets in the application. Thus, our parameters to tune will be the set of all load balancing parameters.

An illustration of this kind of setup is presented in Figure 5.2. The load balancer is placed outside any site for conceptual understanding, but in practice it would exist on all microservice sites to reduce unnecessary network traffic. This would also allow for a more general approach where each individual replica set has its own parameters for load balancing, which could further reduce redundant traffic.

As our model, we adopt the microservice fluid model introduced in Chapter 4. Here, the routing between services is fully determined by the class-to-

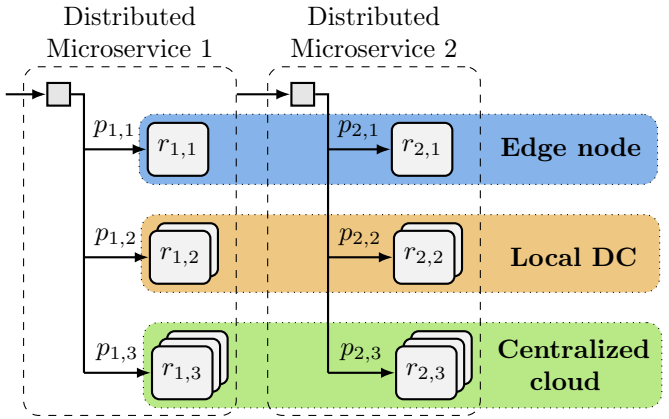


Figure 5.2 An application consisting of two distributed microservices, each with an internal load balancer and replicas $r_{i,j}$ spread over different sites. The control variables are all routing probabilities for the load balancers, where $\boldsymbol{p}_i = [p_{i,1}, p_{i,2}, p_{i,3}]$ are the probabilities for microservice i.

class transitions described in the routing probability matrix $\boldsymbol{P}$. Therefore, it captures the more general load balancing model, where all replicas use a random policy for load balancing requests, with the routing probabilities given by specific rows of $\boldsymbol{P}$. The set of all load balancing routing probabilities is captured in the following definition.

Definition 5.1
Let $\boldsymbol{p}_i$ be a vector of the nonzero probabilities of the row in $\boldsymbol{P}$ corresponding to load balancer i, and let $\mathcal{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \dots\}$ be the set of all load balancing probability vectors in the system.

The load balancing probabilities, along with returning probabilities to multiple downstreams, are the only rows with multiple nonzero values. Therefore, given $\boldsymbol{P}$, the set $\mathcal{P}$ is easy to identify.

The elements of $\mathcal{P}$ constitute our control variables. Changing them will affect the solution of (5.1), we denote this dependence as $\boldsymbol{x}(t \mid \mathcal{P})$.

Obtaining the Desired Performance Metrics

The important performance metrics for the application need to be derived from our model, to form a differentiable mapping from control variables to costs and constraints. We will base the cost of the application on the mean number of requests present in each replica and have a single constraint on the response time percentile, but other costs and constraints obtainable from the fluid model could likewise be considered.

Mean requests in replicas. As each replica is modeled by a single queue, the mean request present at time t can be approximated from the solution of (5.1) by summing over all the phase states present in that queue, i.e.,

$$x_q(t \mid \mathcal{P}) = \sum_{i \in \mathcal{S}_q} x_i(t \mid \mathcal{P}) \qquad q \in \mathcal{Q}. \tag{5.2}$$

Furthermore, let $\boldsymbol{x}_Q \in \mathbb{R}_+^{|\mathcal{Q}|\times 1}$ be the vector of all modeled mean request populations in each queue q in the network.

Response time percentiles. The fluid model can be used to obtain an approximation of the response time CDF, as explained in Section 3.4. Quickly recaptured, assume that we want to obtain the response time over some class subset $\mathcal{C}_R$. Let $\boldsymbol{\pi}(t) \in \mathbb{R}_+^{|\mathcal{S}|\times 1}$ be the probability vector for finding a request in the corresponding phase state after t time units, $\boldsymbol{W}_R = \boldsymbol{\Phi} + \boldsymbol{B}\boldsymbol{P}_R\boldsymbol{A}^T$ where $\boldsymbol{P}_R$ is a reduced version of $\boldsymbol{P}$ where all transition probabilities except between classes in $\mathcal{C}_R$ have been set to 0, and $\beta \in \mathbb{R}_+^{|\mathcal{C}|\times 1}$ a probability vector stating which class in $\mathcal{C}_R$ an arbitrary request will start at $t = 0$. The probability of remaining in $\mathcal{C}_R$ at time t can then be approximated using the following ODE,

$$\frac{d\boldsymbol{\pi}}{dt} = \boldsymbol{W}_R^T D^{g(\boldsymbol{\eta})}\boldsymbol{\pi}(t), \qquad \boldsymbol{\pi}(0) = \boldsymbol{A}\beta(\boldsymbol{\eta}), \tag{5.3}$$

where $D^{g(\boldsymbol{z})} \in \mathbb{R}_+^{|\mathcal{S}|\times|\mathcal{S}|}$ is a diagonal matrix with elements $D_{ii}^{g(\boldsymbol{z})} = g_{Q(i)}(\boldsymbol{\eta})$, i.e., (3.31) evaluated at the stationary solution $\boldsymbol{x}^*$ given some $\boldsymbol{\eta}$. As (5.3) is a linear system, the response time CDF approximation has a closed-form solution as given by (3.48). An approximation of the percentile φ_α can then be obtained by either bisection search over this closed-form solution, or by evaluating (5.3) and finding the t such that $\sum \boldsymbol{\pi}(t = \varphi_\alpha) = 1 - \alpha$. As the percentile and its approximation depend on $\boldsymbol{x}^*$, they also depend on the choice of $\mathcal{P}$, which we denote as $\varphi_\alpha(\mathcal{P})$.

For the considered application, we are interested in the response time of external requests entering the system. In the application model, each external connection has a single recipient class c; therefore, β can be obtained simply by setting $\beta_c = 1$. Furthermore, $\mathcal{C}_R$ will contain all the classes that these external requests can visit before departing the queueing network or reentering the client queue, i.e., the open chain that starts at class c, and ends at the final class of the same queue and request type as explained regarding Remark 4.1.

5.2 Route Optimization Using Automatic Differentiation

The performance metrics retrieved through the fluid model allow us to pose an idealized optimal control version of the cost minimization problem, by

assuming a set of load balancing probability trajectories $\mathcal{P}(t)$ and some cost function $L_o(\cdot)$, as follows:

$$\begin{aligned} \min_{\mathcal{P}(t)} \quad & \int L_o\left[t, \boldsymbol{x}_Q(t), \mathcal{P}(t)\right] dt, \\ \text{subject to } & (5.1), \\ & \sum \boldsymbol{p}(t) = 1 \ \forall t, \boldsymbol{p} \in \mathcal{P}, \\ & \varphi_\alpha\left[\mathcal{P}(t)\right] \leq \varphi_{\text{lim}} \ \forall t. \end{aligned} \tag{5.4}$$

By minimizing over all $\mathcal{P}(t)$ we can try to directly find the optimal load distribution over time that minimizes selected cost. Although feasible, a problem occurs if we try to use this control signal in a real system. The microservice fluid model, extracted from tracing data from a set operating condition, becomes less accurate as $\mathcal{P}(t)$ moves further from its original position. Furthermore, the optimal $\mathcal{P}(t)$ will generally not be convergent, as the underlying cloud system is a dynamic environment.

This requires an optimization scheme where we perform simultaneous online optimization and model tuning in order to both update $\mathcal{P}(t)$ in a robust manner and adapt to changes in the system. To create such an algorithm, some adjustments to (5.4) are needed.

Iterative model refitting & optimization. The model will become less accurate as the operating state changes from what was used to extract the model. Thus, we cannot be certain that a control action based on $\boldsymbol{x}(t)$ in regions far beyond the current state will have the expected effect on the real system. Taking such an action is dangerous, as it can accidentally move the system into operating regions that violate constraints or even yield an unstable system, potentially resulting in application failure.

This can be remedied by continuously updating the model. But due to the fast timescale of the system dynamics compared to the time needed to gather enough data for an accurate model fitting, robust online model tracking at the necessary speed becomes a nontrivial problem. Instead, one possible simple solution is to update $\mathcal{P}$ in discrete steps, where the system is monitored between each step to gather enough data to refit the model before deciding the next $\mathcal{P}_k$. By bounding how far $\mathcal{P}_k$ can move from $\mathcal{P}_{k-1}$, it is then possible to ensure that the system moves within some vicinity of the current operating condition where the model is accurate, increasing the robustness against accidentally violating the constraints. We denote t_k as the time at step k, the sample time as $h = t_k - t_{k-1}$, the data gathered between t_{k-1} and t_k as $\mathcal{H}_k$, and the model refitted on $\mathcal{H}_k$ as $\dot{x} = F_k(\boldsymbol{x})$, where $F_k(\boldsymbol{x}) = \boldsymbol{W}^T\hat{\theta}(\boldsymbol{x}, \boldsymbol{\eta}) + \boldsymbol{A}\boldsymbol{\lambda}$ according to (5.1).

However, this iterative scheme will result in a slower control action, and thus a potentially slower convergence of the cost function minimization than

fully relying on the model to decide some trajectory $\mathcal{P}(t)$ in a single step. In fact, the system will reach a stationary operating condition before deciding the next $\mathcal{P}_k$ as this is required to reliably refit the model. But as the overall goal is to minimize the cost of a running application over a potentially very long time horizon, this slowdown is acceptable as long as the system is not subjected to too many disturbances of too high frequency.

Optimizing over weights. Optimizing directly over the probabilities becomes cumbersome, as we need to adhere to the constraint $\sum \boldsymbol{p} = 1 \; \forall \boldsymbol{p} \in \mathcal{P}_k$. Instead, it is possible to optimize over weight vectors $\boldsymbol{w}$ and enforce the constraint via the *softmax* function $S(\boldsymbol{w})$ [Goodfellow et al., 2016], where

$$S_i(\boldsymbol{w}) = \frac{\exp(w_i)}{\sum_j \exp(w_j)}. \tag{5.5}$$

The softmax function maps vectors defined on $\mathbb{R}^n$ to vectors whose elements are allowed to take values in the interval $[0, 1]$ and that fulfill $\sum_i S_i(\boldsymbol{w}) = 1$. Furthermore, the softmax function preserves the order of the vector element quantities, i.e., if $w_i \geq w_j$, then $S_i(\boldsymbol{w}) \geq S_j(\boldsymbol{w})$. We let each $\boldsymbol{p} \in \mathcal{P}_k$ be associated with a weight vector $\boldsymbol{w}$, and introduce the following.

Definition 5.2
Let $\mathcal{W}_k = \{\boldsymbol{w}_1, \boldsymbol{w}_2, \dots\}$ be the set of all weight vectors at step k, and let $\mathcal{P}(\mathcal{W}_k) := \{S(\boldsymbol{w}) \; \forall \boldsymbol{w} \in \mathcal{W}_k\}$.

To clarify the subscript, the set $\mathcal{W}_k$ will be decided based on the data $\mathcal{H}_k$ collected between steps $k-1$ and k using $\mathcal{W}_{k-1}$.

Limiting the step size. A natural way to manage the step sizes would be to introduce some cost on the difference between $\mathcal{W}_k$ and $\mathcal{W}_{k-1}$. However, certain disturbances such as an increase in the load would increase the overall cost of the system, and thus change the relative step sizes if care is not taken. Instead, we will manage the step size limits by introducing the following constraint on the 2-norm on the change in probability over all weight vectors,

$$\sqrt{\sum_{\boldsymbol{w} \in \mathcal{W}_{k-1}} ||S(\boldsymbol{w}) - S(\boldsymbol{w}^+)||_2^2} \leq d\mathcal{W}_{\text{lim}}, \tag{5.6}$$

where $\boldsymbol{w}^+$ is the corresponding updated $\boldsymbol{w}$ in $\mathcal{W}_k$.

Reworking the percentile constraint. Due to the dynamic nature of cloud systems, we cannot guarantee that any performance metric constraints can actually be fulfilled at any time step. A disturbance might arise that pushes the system to an operating region where a constraint is violated. This can also happen if we are unlucky with the robust stepping of $\mathcal{P}_k$, although we would still be in the vicinity of the constraint limit.

The optimization algorithm must thus be able to handle such cases and quickly drive the system back to a viable operating region. To do this, we can remove the constraint and instead heavily penalize the cost function in the case of violation by, e.g., an additive cost function term $L_\varphi\left[\mathcal{P}\left(\mathcal{W}_k\right)\right]$ using a *penalty function*. The important thing is that the penalization should be negligent as long as the constraint is not violated, sharply increase around the constraint limit, and continue to quickly grow the further from the limit the system moves. This will ensure that the gradient of the cost function points the parameters towards viable operating regions.

A problem can occur in that the response time percentiles obtained from the fluid model are approximative. Hence, the model might indicate that $\hat{\varphi}_\alpha\left[\mathcal{P}\left(\mathcal{W}_k\right)\right]$ is below its limit, while the true value is in fact violated. This can be remedied by basing the *activation* of the penalty function on the estimated φ_α directly from data, while its cost is based on $\hat{\varphi}_\alpha\left[\mathcal{P}\left(\mathcal{W}_k\right)\right]$ to make it differentiable.

New cost function. At time step k, the refitted model $F_k(\boldsymbol{x})$ can then be used to determine the next $\mathcal{W}_k$ using the following new cost function

$$L_k\left(\mathcal{W}\right) = \int_{t_k}^{t_k+t_f} L_q\left[t, \boldsymbol{x}_Q(t), \mathcal{P}\left(\mathcal{W}\right)\right] dt + L_\varphi\left[\mathcal{P}\left(\mathcal{W}\right)\right], \tag{5.7}$$

subjected to $\dot{\boldsymbol{x}} = F_k(\boldsymbol{x})$, where $\boldsymbol{x}\left[t_k \mid \mathcal{P}\left(\mathcal{W}_k\right)\right]$ is set to the stationary solution of $\boldsymbol{x}\left[t \mid \mathcal{P}\left(\mathcal{W}_{k-1}\right)\right]$, and the step size constraint (5.6). As we obtain transient values from the fluid model, we can minimize over these given some arbitrary cost function $L_q(\cdot)$ from current time t_k over some time horizon t_f. In general, as the system will reach a stationary state before the next action is taken, we should let t_f be large enough to capture the stationary values of $\boldsymbol{x}_Q(t)$.

Furthermore, this cost function only takes into account the next step $\mathcal{W}_k$, and there is no prediction horizon for multiple consecutive $\{\mathcal{W}_{k+i}\}_{i\geq 0}$. This is done for simplicity and because we are simply unsure how the model behaves when leaving the vicinity of $\mathcal{W}_{k-1}$. A prediction horizon could potentially be added together with a decreasing trust further away from $\mathcal{W}_{k-1}$, to create an MPC-like problem formulation similar to [Incerto et al., 2017; Incerto et al., 2018b]. This would, however, result in a quite intricate optimization problem with no guarantees on convexity.

Cost Optimizing Algorithm

In each step, we use the cost function (5.7) to decide the next W_k. As no prediction horizon is considered and since the optimization problem is not convex with potentially multiple local minima, we will not try to optimize (5.7) until convergence in each time step. This would become unnecessarily costly, and only result in generating an optimal $\mathcal{W}_k$ given the step length

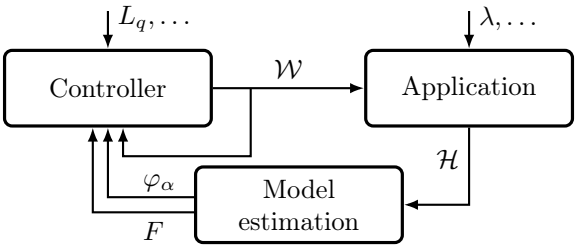


Figure 5.3 An illustrative block diagram of the cost optimizing algorithm. Data $\mathcal{H}$ is collected from the application and used to fit model F as well as estimate the response time φ_α. These are then used by the controller to decide $\mathcal{W}$ based on the old weights and the gradient of the cost from the estimated model and percentile. Disturbances can act on both the controller, in the form of, e.g., changes in the cost L_q, and on the application, in the form of, e.g., changing workload λ.

constraint with no guarantees that it would actually move the system towards its global minimum.

Instead, a more direct approach is suitable. For demonstration purposes, we use a very simple single gradient step to decide $\mathcal{W}_k$, based on the gradient of (5.7). Using automatic differentiation, this gradient ∇L_k can be obtained directly, despite dependence on the two ODEs (5.1) and (5.3). Together with some constant scaling factor δ and a step size limiter c, we can then calculate the next $\mathcal{W}_k$ with the following gradient step update

$$\boldsymbol{w}^+ = \boldsymbol{w} - c\delta\nabla_{\boldsymbol{w}} L_k\left(\mathcal{W}_{k-1}\right) \qquad \forall \boldsymbol{w} \in \mathcal{W}_{k-1}, \tag{5.8}$$

where $\nabla_{\boldsymbol{w}}$ is the gradient w.r.t. the elements in $\boldsymbol{w}$, and $\boldsymbol{w}^+$ the corresponding weight vector in $\mathcal{W}_k$. If (5.6) is not violated, then $c = 1$, otherwise it can be obtained via, e.g, bisection search to find a $c \in [0, 1]$ such that

$$\sqrt{\sum_{\boldsymbol{w}\in\mathcal{W}_{k-1}} \left\|S(\boldsymbol{w}) - S\left[\boldsymbol{w}^+(c)\right]\right\|_2^2} = d\mathcal{W}_{\text{lim}}. \tag{5.9}$$

The complete algorithm can thus be seen as an adaptable *gradient descent* scheme, where after each step we reevaluate the model before calculating the gradient and deciding the next step to take. The algorithm is illustrated in the block diagram in Figure 5.3, and summarized by the pseudocode in Algorithm 1.

Algorithm 1 Cost optimizing algorithm.

Algorithm 1a The control loop; Run N iterations of data collection and parameter updates. Data collection is run for a duration of h t.u.

Initialize $\mathcal{W}_0$
for $k \leftarrow 1$ to N **do**
  Set $\mathcal{P}(\mathcal{W}_{k-1})$ as load balancing strategy
  $\mathcal{H}_k \leftarrow \text{COLLECT\_DATA}(h)$
  $F_k, \varphi_\alpha \leftarrow \text{FIT}(\mathcal{H}_k)$
  $\mathcal{W}_k \leftarrow \text{UPDATE}(\mathcal{W}_{k-1}, F_k, \varphi_\alpha)$
end for

Algorithm 1b The controller; Calculate $\mathcal{W}_k$ based on $\mathcal{W}_{k-1}$ and estimated model F_k and percentile φ_α.

function $\text{UPDATE}(\mathcal{W}_{k-1}, F_k, \varphi_\alpha)$
  $\mathcal{W}_k \leftarrow \emptyset$
  for $\boldsymbol{w} \in \mathcal{W}_{k-1}$ **do**
    $\nabla_{\boldsymbol{w}} L_k \leftarrow \text{GRADIENT}(L_k(\boldsymbol{w} \mid F_k, \varphi_\alpha), \boldsymbol{w})$
    $\boldsymbol{w}^+ \leftarrow \text{LIMITED\_STEP}(\boldsymbol{w}, \nabla_{\boldsymbol{w}} L_k)$
    $\mathcal{W}_k \leftarrow \mathcal{W}_k \cup \{\boldsymbol{w}^+\}$
  end for
  return $\mathcal{W}_k$
end function

5.3 Experimental Evaluation

To test and showcase the cost optimizing algorithm, three experiments were performed on the small microservice application from Chapter 4.

Experimental Setup

A similar setup to the one used in Section 4.4 was considered, consisting of the simple facedetection-as-a-service application deployed on the FedApp sandbox. In total, three clusters were deployed with 4 virtual machines each, where each VM had 4 vCPU and 4 Gb of RAM.

The application. Quickly recapped, the face detection example application has 3 services; one `frontend`, one `backend` and one `storage`. Users can interact with the application via two endpoints on the `frontend`, `detect/` and `fetch/`. The `frontend/detect/` endpoint accepts requests with an attached image, that is preprocessed and then forwarded to `backend/detect/`. This service endpoint then performs the face detection and sends the detected image to `storage/store/` before returning it to the caller via the `frontend`. Similarly, the `frontend/fetch/` endpoint accepts requests with an attached

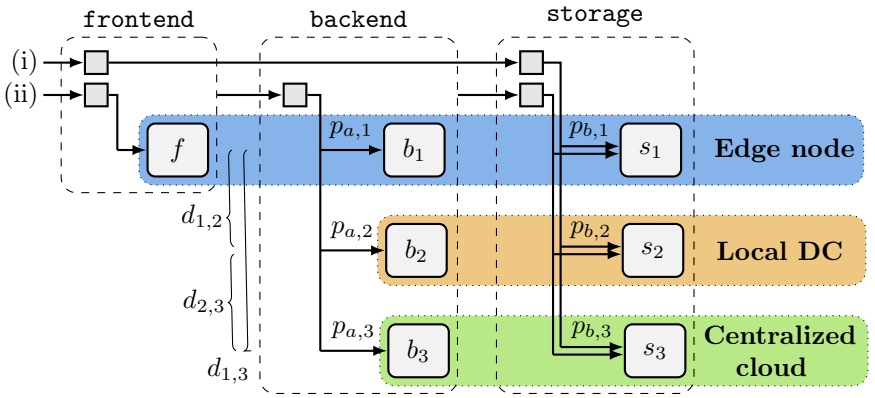


Figure 5.4 The incoming load passes through the `frontend` at either the (i) `fetch/` or (ii) `detect/` endpoint and is, respectively, balanced over either the `backend` or `storage` replicas according to the associated probability vector $\boldsymbol{p}_m$. The delay between cluster i and j are given by $d_{i,j}$, which is assumed to be symmetric. The `backend` and `storage` replicas with lower delays to the `frontend` are assumed to be more expensive to run.

image name, that is forwarded to `storage/fetch/`. This service endpoint then returns the image with that name to the caller via the `frontend`.

We deployed this application on 3 clusters using FedApp as shown in Figure 5.4. The user facing `frontend` has one replica f on the edge close to its users, while the `backend` and `storage` has three replicas b_i, s_i distributed across the three sites, emulated by our different clusters. Each cross-cluster connection was associated with a delay $d_{i,j}$, and each b_i or s_i was associated with a computation cost C_i. We assumed that higher costs were associated with lower delays, in order to create a trade-off between cost and latency. Such scenarios could occur in, e.g., fog/edge computing, where low latency computations can be performed on the device or at geographically close edge data centers but at an increased cost, while offloading computations to larger sites is cheaper but subjected to longer communication delays. The more general load balancing model was assumed, where each replica has its own set of load balancing probabilities for routing outgoing requests. Hence, as we have one `frontend` sending requests to both $b_{1:3}$ and $s_{1:3}$, it will have two load balancing probability vectors $\boldsymbol{p}^{f,b}, \boldsymbol{p}^{f,s}$ with 3 elements each. Furthermore, as we have b_i $i \in 1:3$, each sending requests to $s_{1:3}$, each b_i will have a $\boldsymbol{p}^{b_i,s}$ with 3 elements. In total we thus obtain $|\mathcal{P}| = 5$ load balancing vectors and a total of 15 parameters to optimize over.

Before experimenting on the whole application, we first studied a simpli-

fied version consisting of only two services, the `frontend` and the `backend`. This simple application was deployed in a similar manner to Figure 5.4 with one `frontend` replica close to the user at the edge and `backend` replicas on each cluster. This setup yields only a single application endpoint at `frontend/detect/` and a single load balancing probability vector.

Algorithm implementation. Using a load generator, images were fed to the `frontend/detect/` endpoint as Poisson arrivals with rate λ. When considering the full application $N = 50$ clients were also created to fetch images via the `frontend/fetch/` endpoint with exponentially distributed waiting times with mean $1/\mu_c$. At each time step k, tracing data from Istio was collected to generate $\mathcal{H}_k$, which was then used to fit the model F_k to $\mathcal{H}_k$ and decide the next set of parameters $\mathcal{W}_k$.

The model fit, cost function, and gradient stepping were implemented using the Julia[1] programming language, which has a broad library of different methods for automatic differentiation of Julia-native functions. We used `ForwardDiff.jl` [Revels et al., 2016] together with the ODE solver package `DifferentialEquations.jl` [Rackauckas and Nie, 2017], which allows for automatic differentiation of the cost function, enabling the gradient ∇L_k and the next $\mathcal{W}_k$ to be effortlessly calculated. The application and our implementation of the cost optimizing algorithm is available on GitHub[2].

Two Backends – Fixed Steps in an Offline Experiment

In a first experiment, we considered the simplified example application with only two services, and further only two replicas of the `backend`, b_1 deployed on the same cluster as the `frontend`, and b_2 deployed on a different cluster. All connections between the two clusters were given a Pareto distributed additive delay with a 25 ms mean, 5 ms *jitter* (a TC netem term roughly equating standard deviation) and 25% correlation between samples. Hence, requests that were load balanced to b_2 experienced an additive delay.

The probability constraint enables us to determine the load balancing probabilities directly by using a single parameter p_1, and then determining $p_2 = 1 - p_1$, removing the need for the weights and softmax function. This makes it feasible to collect data in a grid over $p_1 \in [0, 1]$, and in turn allows us to run the cost optimizing algorithm against the recorded data with a fixed step size over this grid.

We let the cost function be conditional on φ_α violating φ_{lim} for selecting either L_φ or L_q. L_φ was set to a scaled $\hat{\varphi}_\alpha(p_1)$, whereas L_q was set to 0 everywhere except at time t_f where it was a linear function of the state. The

[1] https://julialang.org/
[2] https://github.com/JohanRuuskanen/ACSOS2022_code/tree/thesis

stationary cost in L_q was chosen to make it easier to compare to data.

$$L_k(p_1) = \begin{cases} C_\varphi \hat{\varphi}_\alpha(p_1) & \text{if } \varphi_\alpha > \varphi_{\text{lim}}, \\ \boldsymbol{C}^T \boldsymbol{x}_Q(t_k + t_f \mid p_1) & \text{otherwise .} \end{cases} \tag{5.10}$$

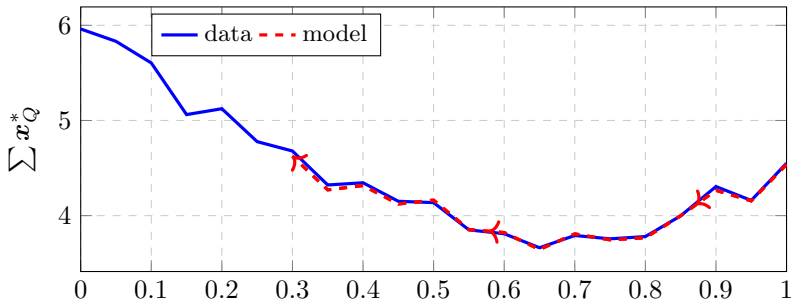
The conditional uses recorded data, as we would like the constraint to be active if the estimated φ_α from data violates the constraint. The actual cost was then implemented using the model prediction $\hat{\varphi}_\alpha(p_1)$ to make it differentiable. We let $\alpha = 0.95$ to consider the 95th percentile of response times to the application, $\varphi_{\text{lim}} = 0.55$ s, $t_f = 5$ s, and $\boldsymbol{C}$ be a zero vector, except for two `backend` replicas where it was set to $\boldsymbol{C}_b = [3, 1]$. We set $C_\varphi = 8$, although in this case it only affects the plot scaling, since the cost is conditional and the gradient step is fixed. For the model, we gave each class in the queueing network 3 phase states, for a total size of $|\mathcal{S}| = 24$.

Data for the offline experiment was recorded by creating a grid over p_1 with steps of 0.05 between 0 and 1. For each value of p_1, data was recorded for $h = 300$ s using an arrival rate $\lambda = 14$. The cost optimizing algorithm was then run against this recorded data, starting from $p_1 = 1$ and stepping along the grid in the gradient direction. The results can be seen in Figure 5.5. Figure 5.5(a) shows the mean total requests present in the application, Figure 5.5(b) shows the percentiles, and Figure 5.5(c) shows the cost based on (5.10). The apparent noisiness can be attributed to the noisy data used for refitting the model.

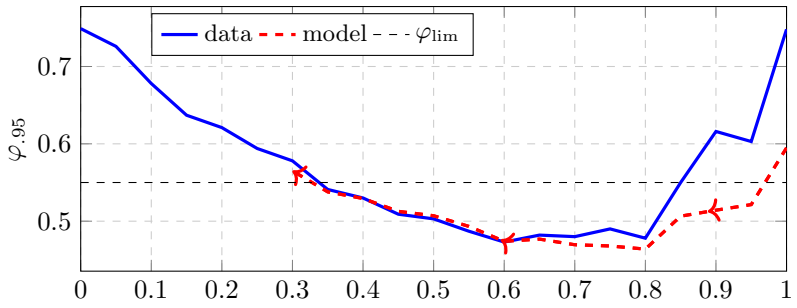
As can be seen, the algorithm manages to step in the direction of decreasing cost and ultimately find the minimum. Starting at $p_1 = 1$, the system is passing all load to b_1 resulting in $\varphi_{.95} > \varphi_{\text{lim}}$ and the cost is then based on the response time percentile. Moving in the direction of $-d\varphi_{.95}/dp_1$, the percentile and the total queue length are decreasing, as we offload b_1 by routing some load to b_2 instead. When crossing the threshold φ_{lim}, the cost switches to L_q. Even though we see that the total queue length starts to grow as p_1 goes below 0.65, the cost for b_1 is higher than b_2, and thus the queue-based cost is still decreasing with decreasing p_1. However, before reaching $p_1 = 0$, we hit the φ_{lim} threshold again, and here the simulation is stopped since the algorithm will start to alternate between fixed steps in different directions.

Three Backends – Online Optimization Experiment

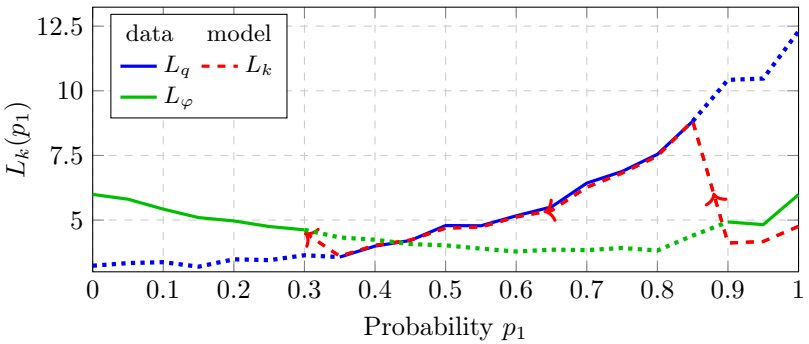
For the second experiment, the entire cost optimizing algorithm was run live on the simplified example application. We considered the same setup as in the first experiment, but also introduced the third `backend` replica b_3 deployed on the third cluster. All connections between the first and third clusters were given a Pareto distributed additive delay with a 50 ms mean, 10 ms jitter, and 25% correlation between samples. As we now have more than 1 parameter to



(a) Total mean requests in application.



(b) 95th percentile of response time.



(c) Partial and full cost.

Figure 5.5 Results from the offline experiment with two `backend` replicas. The values are plotted over the probability p_1. In 5.5(a) and 5.5(b) the values from recorded data $\mathcal{H}$ (blue line) are compared to the corresponding model fit (red dashed) In 5.5(b) φ_{lim} also shows when the cost (5.10) switches mode. The cost in turn can be seen in 5.5(c) for the cost of the queues (blue) and the cost of violating the response time threshold (green). The lines are filled where each cost is active, and dotted where they are inactive. The red dashed line shows the model's estimate of the full cost.

optimize over, we resorted to using the weight vectors in $\mathcal{W}$ as described in Section 5.2.

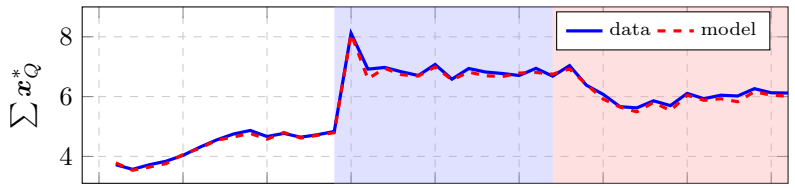
For the cost function, we again let L_q be a linear function of the queue lengths at t_f and 0 at other times, only to consider stationary values for simplifying comparisons with data. However, the response time constraint was included by setting L_φ as an exponential penalty function based on the difference between φ_α and φ_{lim}. Again, we based the activation of the constraint on recorded data, as we would like it to be active when the real φ_α is near or above its limit. The resulting cost function becomes

$$\begin{aligned} L_k(\mathcal{W}) = \boldsymbol{C}^T \boldsymbol{x}_Q \left[t_k + t_f \mid \mathcal{P}\left(\mathcal{W}\right)\right] \\ + C_\varphi e^{\nu(\varphi_\alpha - \varphi_{\text{lim}})} \hat{\varphi}_\alpha \left[\mathcal{P}\left(\mathcal{W}\right)\right], \end{aligned} \tag{5.11}$$

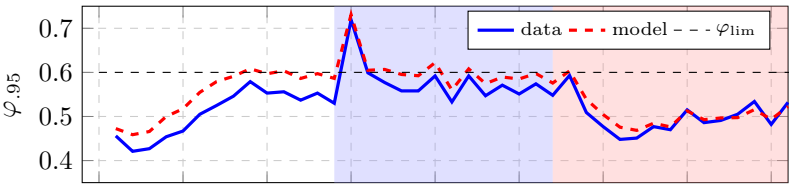
where we set $\alpha = 0.95$, $\varphi_{\text{lim}} = 0.6$ s, $\nu = 10$, $C_\varphi = 5$, $t_f = 5$ s and $\boldsymbol{C}$ to a zero vector except for the `backend` replicas where it was set to $\boldsymbol{C}_b = [3, 2, 1]$. Furthermore, the gradient step parameters were set to $\delta = 0.5$ and $d\mathcal{W}_{\text{lim}} = 0.15$. For the queueing network model, we gave each class 3 phase states for a total of $|\mathcal{S}| = 33$.

The system was loaded with Poisson arrivals of rate $\lambda = 15$ for a total of 40 time steps, with the initial weight vector set to $\boldsymbol{w}_0 = [2, 0, 0]$, giving $\boldsymbol{p}_0 \approx [0.78, 0.11, 0.11]$. Each time step was given a duration of $h = 300$ s. We further let the running system be influenced by two disturbances. The first disturbance was introduced at time step 14, where the arrival rate suddenly increases by 50% to $\lambda = 22.5$. The second disturbance was introduced at time step 27, where the costs for the `backend` replicas are changed to $\boldsymbol{C}_b = [1, 2, 3]$. The results of this experiment can be seen in Figure 5.6, showing the total mean requests present in the application, the response time percentiles and the cost based on (5.11) over the time steps. Furthermore, Figure 5.6(d) shows the three load balancing probabilities from the `frontend` to b_1 (p_1, blue), to b_2 (p_2, red dashed) and to b_3 (p_3, green dotted).

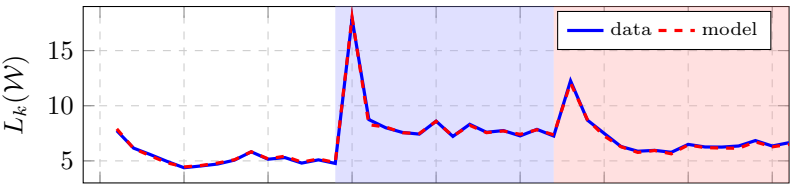
As can be seen, the online optimization algorithm manages to drive the system towards a load balancing setting of less cost and counteract the disturbances in a few steps. At first, the system shifts load from b_1 to b_2 and b_3. This decreases the cost, but it also increases the total queue length and response time percentile, as b_2 and b_3 are associated with a greater site-to-site delay. The shift is mostly stopped when the percentile constraint is reached, but due to the simplicity of the gradient descent approach, the system experiences a slow final convergence. When the first disturbance in the form of a 50% increase in load is introduced, both the queue length and percentiles increase immediately. As the percentile constraint is now violated, the cost function spikes, resulting in the gradient step aggressively moving the system back into a parameter configuration where the constraint is no longer violated. After the constraint is fulfilled once again, the cost function slowly



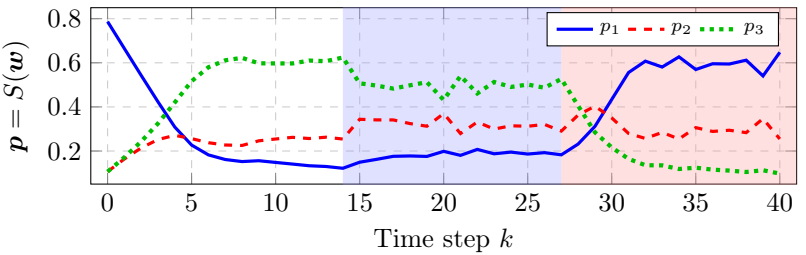
(a) Total mean requests in the application.



(b) 95th percentile of the response times.



(c) Cost function.



(d) Load balancing probabilities.

Figure 5.6 Results from the online experiment with three `backend` replicas. The values are plotted over the time steps, each corresponding to a sampling period of 300 seconds. In 5.6(a), 5.6(b) and 5.6(c) values from recorded data $\mathcal{H}$ (blue line) are compared to the corresponding model fit (red dashed). Further, in 5.6(d) the three lines corresponds to the three load balancing probabilities. Finally, the blue shaded area shows where the first disturbance is active on the arrival rate, while the shaded red area shows where the first and second disturbance are active on both arrival rate and queue length costs. A one time step lag can be seen on 5.6(a), 5.6(b), 5.6(c) compared to 5.6(d) as $\mathcal{H}_k$ is recorded using $\mathcal{W}_{k-1}$.

settles to a minimum. At the activation of the second disturbance, in the form of shifting the costs of b_1 and b_3, the cost function increases while the queue length and percentile remain the same. The system then quickly shifts the load probabilities and reduces the cost until the effects of the penalty function become too large and settles to a slow final convergence.

At a few steps, it seems that the algorithm steps upwards in cost, but this can be attributed to noise. Furthermore, as can be seen in Figure 5.6(b), there are at times rather large gaps remaining between the percentile and its limit. This has to do with the values assigned to the penalty function. Increasing C_φ and α would lead to more aggressive handling of violations and a tighter gap, but they were kept fairly low to yield a more presentable cost function.

Full Application Experiment

For the third and final experiment, we considered optimization at runtime of the entire example application deployment shown in Figure 5.4. As we here have 5 load balancing probability vectors in $\mathcal{P}$, for a total of 15 parameters, we again resorted to optimizing over the weight vectors in $\mathcal{W}$.

Since we load two endpoints with request arrivals, each endpoint was given a separate percentile constraint on its response times. A similar cost function to (5.11) was used. The queue length cost L_q was again set to a linear function of the queue lengths at time t_f, and the two response time constraints were added as separate exponential penalty functions $L_\varphi^{\texttt{detect/}}, L_\varphi^{\texttt{fetch/}}$. The activation of these penalty functions was again based on recorded data. The resulting cost function becomes

$$\begin{aligned} L_k(\mathcal{W}) = \boldsymbol{C}^T \boldsymbol{x}_Q \left[t_k + t_f \mid \mathcal{P}\left(\mathcal{W}\right)\right] \\ + C_\varphi^o e^{\nu^o(\varphi_\alpha^o - \varphi_{\lim}^o)} \hat{\varphi}_\alpha^o \left[\mathcal{P}\left(\mathcal{W}\right)\right] \\ + C_\varphi^c e^{\nu^c(\varphi_\alpha^c - \varphi_{\lim}^c)} \hat{\varphi}_\alpha^c \left[\mathcal{P}\left(\mathcal{W}\right)\right], \end{aligned} \tag{5.12}$$

where $o = \texttt{detect/}$ and $c = \texttt{fetch/}$. Furthermore, we set $\alpha = 0.95$, $t_f = 5$ s, and C to a zero vector except for the `backend` and `storage` replicas, where it was set to $\boldsymbol{C}_b = [6, 4, 1]$, $\boldsymbol{C}_s = [6, 4, 1]$. The two response time percentile limits were set to $\varphi_{\lim}^{\texttt{detect/}} = 1.25$ and $\varphi_{\lim}^{\texttt{fetch/}} = 0.4$, while the parameters for the penalty function where set to $C_\varphi^{\texttt{detect/}} = C_\varphi^{\texttt{fetch/}} = 100$, $\nu^{\texttt{detect/}} = 12$, $\nu^{\texttt{fetch/}} = 30$. For the queueing network model, we gave each class 3 phase states for a total of $|\mathcal{S}| = 139$.

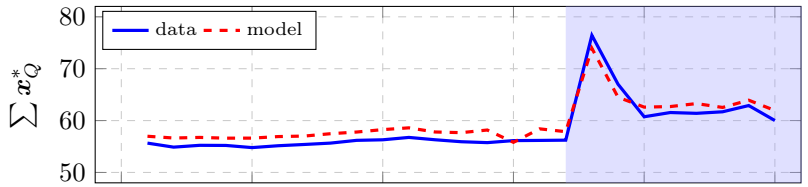
The system was then loaded for a total of 25 time steps, each given a duration of $h = 300$ s. The Poisson arrivals to the `detect/` endpoint were given the rate $\lambda = 15$ and the clients connecting to the `fetch/` endpoint were given the waiting time $\frac{1}{\mu_c} = 1.0$. Initial values for the `frontend` replica were set to $\boldsymbol{w}_0^{f,b} = \boldsymbol{w}_0^{f,s} = [2, 0, 0]$, prioritizing sending requests to b_1 and

s_1 on the first cluster. For the `backend` replicas, they were instead set to $\boldsymbol{w}_0^{b_1,s} = \boldsymbol{w}_0^{b_2,s} = \boldsymbol{w}_0^{b_3,s} = [0, 0, 0]$ loading the `storage` replicas equally across the clusters. At $k = 17$ the system was subjected to a load disturbance, where the Poisson arrival rate was increased by 50% to $\lambda = 22.5$ and the waiting times for the clients were reduced by 50% to $\frac{1}{\mu_c} = 0.5$. The results can be seen in Figure 5.7, showing the total mean requests present in the application, the response time percentiles for both endpoints and the cost based on (5.12) over the time steps. Furthermore, Figure 5.8 shows the load balancing probabilities over the time steps.

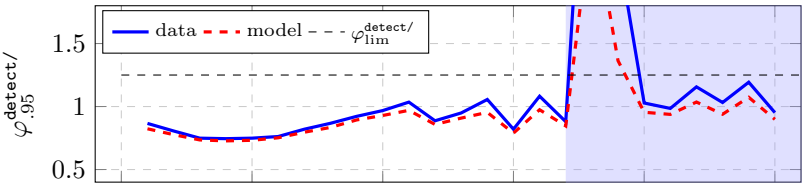
As can be seen, the optimization algorithm manage to drive the queue length costs down, despite having to simultaneously tune the 15 load balancing parameters. At first, it seems that the algorithm prioritizes tuning the `frontend` load balancing parameters, which decreases costs at the expense increasing the percentiles and slightly the total mean requests. After about 8 time steps, the response time percentiles, especially $\varphi_{.95}^{\texttt{fetch}}$ starts to get close to its limit, and the algorithm seems to shift its focus to tuning the `backend` load balancing parameters, which further decreases the costs. After the workload disturbance both response time percentile constraints becomes violated, but the algorithm manages return to an valid operating condition after a few steps by focusing on retuning the `frontend` load balancing parameters.

In this experiment, some downsides of using our simple gradient stepping optimization scheme can be observed. First, looking closer at Figure 5.7(d), it can be seen that the algorithm after a few steps actually starts to increase the cost function that we are trying to minimize. The upward trend is split between alternating increases and decreases in cost. Hence, this can probably be attributed to the algorithm stepping back and forth along the edges of the penalty functions. When a penalty function is small but close to its limit, naively stepping in the direction of the cost function negative gradient might yield parameters where the penalty function is much larger. The new gradient at these parameters can in turn point in a new direction that after another step yields a higher cost than the parameters two steps back and so on. This behavior is not desired, but the algorithm still manages to reduce the queue-length cost, which is what we actually care about.

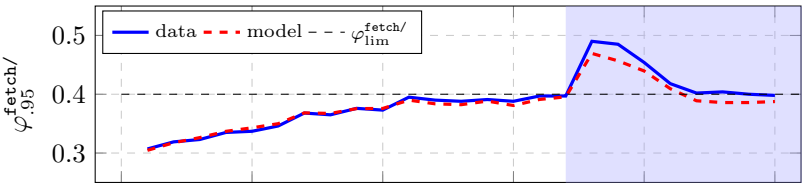
Furthermore, after the disturbance has been introduced, we can in Figure 5.8(a) see that the algorithm seems to deal with the constraint violations one at a time. A probable cause for this is that the exponential penalty function increases very quickly in cost when its limit is violated. Therefore, if more than one penalty function is simultaneously violated, one of the functions, in this case $L_{\varphi}^{\texttt{detect/}}$, might get a much higher value than the rest. Reducing this function then becomes prioritized in the gradient step and will lead to a slower return to a valid operating condition where no constraints are violated.



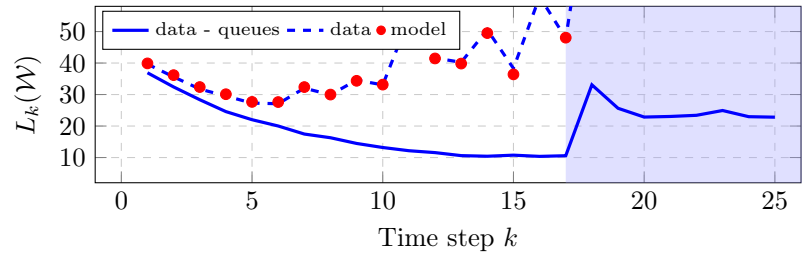
(a) Total mean requests in the application.



(b) Response time p95 concerning the `detect/` endpoint.

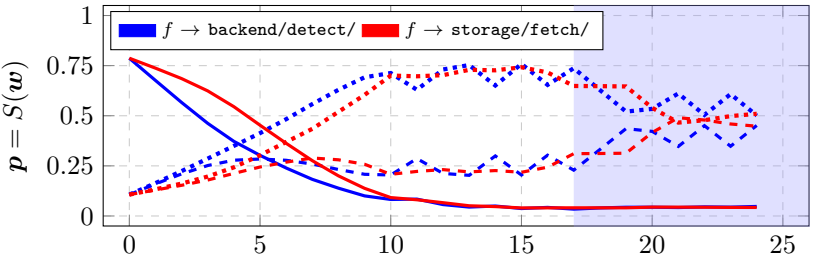


(c) Response time p95 concerning the `fetch/` endpoint.

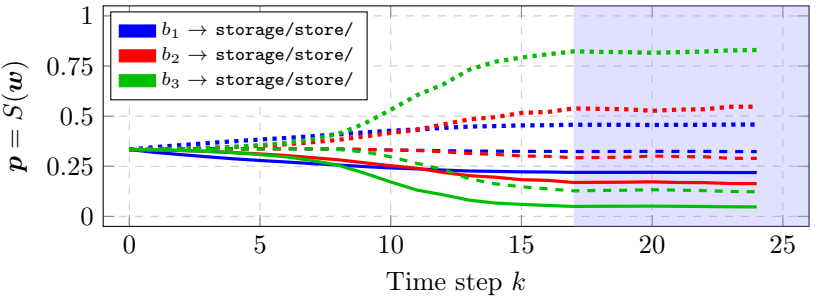


(d) Cost function.

Figure 5.7 Results from the online experiment with the complete face detect application. The values are plotted over the time steps, each corresponding to a sampling period of 300 seconds. In all subfigures, values from recorded data $\mathcal{H}$ (blue) are compared to the corresponding model fit (red). In 5.7(d), both queue length cost L_q (blue line) and the total cost (blue dashed) are shown. The blue shaded area shows when the workload disturbance is active.



(a) `Frontend` load balancing probabilities.



(b) `Backend` load balancing probabilities.

Figure 5.8 Load balancing probability parameters from the online experiment with the complete face detection application. The full lines shows the probability to send to a replica in cluster 1, the dashed lines to a replica in cluster 2 and the dotted lines to a replica in cluster 3. In 5.8(a) the probabilities between the single `frontend` replica on cluster 1 to the three `backend` replicas (blue) and storage replicas (red) are shown. In 5.8(b) the load balancing probabilities from the `backend` replicas on cluster 1 (blue), cluster 2 (red) and cluster 3 (green) to the three `storage` replicas are shown. The blue shaded area shows when the workload disturbance is active.

Comment on Evaluation Speed

For the simplified application with three `backend` replicas, the model fitting and parameter update according to Algorithm 1 take around 15 s. The majority of the time ($\approx$ 13 s) goes towards fitting the 11 PH distributions using the EM algorithm, which can be made faster by, e.g., moment matching [Osogami and Harchol-Balter, 2006]. The differentiation step itself is quick, taking around 180 ms. However, on the full application, model fitting and parameter update takes around 80 s, with updating the 47 PH distributions taking 60 s, and the differentiation step noticeably slower at 4.7 s.

The forward-mode automatic differentiation using `ForwardDiff.jl` scales as $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{P}|)$, i.e., the number of states times the number of parameters [Revels et al., 2016]. Hence, the evaluation of the gradient should be some-

where in the vicinity of $(139 \cdot 15)/(33 \cdot 3) \approx 21$ times slower for the full application, which is about right as $4.7/0.18 \approx 26$. This might seem slow, but taking into account that evaluating the cost function takes approximately 110 ms and 170 ms for the simplified and standard application, respectively, it is still faster than estimating the gradient with a naive finite difference. This would roughly take for the simplified application $3 \cdot 2 \cdot 110 = 660$ ms and for the full application $15 \cdot 2 \cdot 0.17 = 5.1$ s. However, a finite difference approach is not constrained to Julia-native ODE solvers, which could potentially be slower than other more optimized alternatives.

5.4 Summary and Discussion

In this chapter, we have demonstrated how automatic differentiation can be used to optimize a running distributed microservice application. This is done by deriving an online optimization scheme to minimize some holistic cost by tuning the probabilities of random load balancers between replica sets. Although it is not guaranteed to find the global cost minimum, the algorithm is shown to reduce cost while adhering to constraints on the response time percentile in experimental evaluations. As the assumed microservice fluid model is fairly general and the cost function and its constraints can be arbitrarily defined, this online optimization scheme can be quickly adapted for a multitude of different load balancing scenarios.

Discussion. Automatic differentiation allows us to obtain the derivatives of functions that would be too difficult to explicitly derive. In our case, it is used to differentiate through an arbitrary cost function L based on the solution of two dependent ODEs from our fluid model. Other models, in addition to our adapted microservice fluid model, can potentially be considered for this purpose. However, as this model can be quickly extracted online from data, has expressions for response time percentiles, and is relatively quick to evaluate and differentiate using the Julia ecosystem, it can be used to derive an online cost optimizing algorithm that adheres to percentile constraints.

The slow evaluation of our current implementation of the cost optimizing algorithm is a hindrance for tackling larger systems. For these cases, time needs to be spent on optimizing the code. Focus should be put on faster methods for obtaining the PH distributions and faster evaluation of the cost function. Furthermore, due to the poor scaling of forward-mode automatic differentiation in the number of parameters and states, other methods for automatic differentiation with better scaling properties should be investigated, see, e.g., [Rackauckas et al., 2020, Section 6.3].

The generality of our cost optimizing algorithm is not only a boon but also gives the approach some drawbacks. First, there are no convergence guarantees, and the performance is inherently dependent on the accuracy of

the model. Identifying when a model is performing poorly is thus important to avoid driving the system into regions of high cost or constraint violations. In addition, deriving a suitable cost function and good optimization parameters will require both time and expert knowledge.

Finally, the simplicity of using a single gradient step to update the parameters can become problematic for more advanced systems. The algorithm risks getting stuck in local minima, and as seen in the full application experiment, it can potentially also lead to an increase of the total cost. For this particular experiment, the algorithm still manages to decrease the important queue length costs, but the question remains if this generalizes to all settings. We recon that it probably should for most cases, as the gradient step will always try to move in the direction of decreasing queue length cost as long as the penalty functions are small.

6

Modeling Request Cloning Using Synchronized Service

Request cloning offers a possibility to significantly improve the performance of cloud applications that experience large uncertainties in the computation time of its requests [Ananthanarayanan et al., 2013]. By sending copies of user requests to multiple instances of the same server, both the response time mean and its tail could potentially be reduced. However, this also introduces extra load on the system which instead risk increasing the response time. Thus, to determine the particular benefits and a suitable number of clones, it is important that the considered cloning system can be adequately modeled.

Introduction

Previous modeling research for request cloning and speculative execution has mainly focused on queueing theory with specific IID interarrival and service time distributions under the FCFS discipline. In this chapter, we however take another approach by introducing a concept we refer to as *synchronized service*, that allows certain cloning systems to be equivalently represented as a single G/G/k queue. This representation requires no assumptions on the distributions for the involved queues, and is valid under any deterministic queueing discipline and number of processors k, as long as they are the same across all queues.

The equivalent G/G/k model implies that if there exists methods for extracting performance metrics for the resulting queue, then these can be used to analyze the entire cloning system. This is used to study the optimal number of clones over sets of M/G/1-PS queues, both for the baseline synchronized system, and for sets of synchronized systems behind a load balancer. Since synchronized service requires the introduction of unrealistic assumptions, we further study what happens to synchronized systems if these assumptions are relaxed. For this we limit ourselves to the PS discipline, as it is prevalent

in modeling cloud applications and has not previously been studied in conjunction with request cloning. Finally, we introduce a method based on the equivalent G/G/k model that can approximately model speculative execution in systems with server-side queueing.

The theoretical findings are validated using a discrete event simulator, and it is shown that we in accordance with theory are able to precisely predict the behavior of server systems subject to cloning that fulfill the assumptions for synchronized service. Further, it is shown that for cloning over PS queues, relaxing the necessary assumptions only has a minor effect on the mean response time under low loads or certain load balancers such as JSQ. For these cases, the equivalent G/G/k model serves as an accurate approximation.

Outline. This chapter is structured as follows. In the end of this introduction, some necessary modeling assumptions and the cloning simulator used for our experiments are introduced. In Section 6.1 we then introduce the concept of synchronized service, the necessary assumptions and the equivalent G/G/k model that arises for such systems. In Section 6.2, the usage of the model is exemplified by studying the optimal number of clones with respect to mean response time. Further, in Section 6.3 we study what happens if the assumptions necessary for synchronized service are relaxed, and in Section 6.4 extend the model to allow for speculative execution. Finally, the chapter is summarized in Section 6.5.

Assumptions and Notations

We assume a system of n servers, subjected to original requests r^{orig} from some arrival process. Each original request is then copied into a set of clones $\mathcal{R}^c$, and dispatched across the n servers in some manner. The first clone in this set to complete its service becomes the response of its r^{orig}. The rest of the clones are assumed to be subsequently canceled according to the *Cancel-on-Complete* (CoC) principle. This is necessary for synchronized service, and the PS discipline which we will later focus on is not compatible with *Cancel-on-Start* (CoS) cloning as every clone starts its service immediately upon arrival. Henceforth, we will assume that cloning is synonymous with CoC cloning, and refer a system of the type introduced above as a *cloning system*.

Each server will be modeled with a single queue with k internal servers, to avoid confusion we will refer to these as *processors* in this chapter. Let $\Phi_i^s(t)$ denote the CDF of the service time distribution for server i, and $\Phi^\lambda(t)$ denote the CDF of the interarrival time distribution between requests. At first, no assumptions on either $\Phi_i^s(t)$, $\Phi^\lambda(t)$ nor queueing disciplines are made, apart from that the queueing discipline is deterministic and the same across all servers. To enable further analysis, we will later restrict our assumptions and require $k = 1$, the PS discipline and homogeneous service times.

The Cloning Simulator

In this chapter, we demonstrate and evaluate the to-be-stated examples and claims using our own discrete event simulator[1]. We refrained from using existing simulators like CloudSim [Calheiros et al., 2011] or real experiments, both due to simplicity and since we wanted to evaluate our results without having to take modeling errors between the (simulated) infrastructure and queueing models into consideration.

To construct the simulator, we took inspiration from the brownout simulator[2] (see [Klein et al., 2014]). Modifications were made to remove the adaptation layer and add cloning functionality. The resulting cloning simulator includes the options to define: (i) the interarrival time distribution and the service time distributions individually for the n servers, (ii) the number of clones or *cloning factor* c_f, (iii) the load balancing strategy and (iv) arrival and cancellation delays. The arrival and service time distributions can be heterogeneous and even allowed to be set based on empirical CDF data.

6.1 Synchronized Service and the G/G/k Model

We define synchronized service of a cloned request as follows

Definition 6.1—Synchronized service
An original request r^{orig} is said to receive *synchronized service* if all its clones $\mathcal{R}^c$ begins and ends their service at the same time.

To show when this concept holds, the following two central assumption regarding the number and placement of the clones to every r^{orig}, and the absence of timing issues between arrivals and cancellations of clones, will first be introduced.

Assumption 6.1—Clone-to-all
Every original request r^{orig} is cloned once to all available servers $s_{1:n}$. Denote this set of clones as $\mathcal{R}^c := r^c_{1:n}$, signifying that clone r^c_i is placed on s_i.

Assumption 6.2—Synchronous arrivals and cancellations
All clones of every r^{orig} arrive at their respective servers at the same time, and after the completion of the service of the first clone to r^{orig}, all its clones are removed from service at the same time.

Note that completed clones to r^{orig} are assumed to still occupy processors until all clones are simultaneously removed. Further, let $a \geq 0$ be the time between arrival of r^{orig} to the cloning system and arrival of $r^c_{1:n}$ to the servers,

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

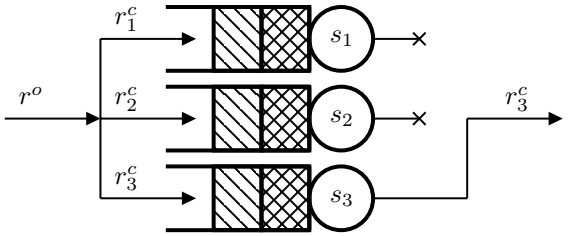


Figure 6.1 Clone-to-all: A cloning system of 3 servers under synchronized service. Each server is represented with a single queue. Each arriving request r^{orig} is cloned to the 3 servers, and by the assumption of synchronized arrival all clones r_1^c, r_2^c, r_3^c will enter their servers simultaneously. The response to the client is produced by the server that completes its cloned request in the smallest time (for this particular r^{orig} by server 3). Given CoC cloning and the assumption of synchronized cancellation, the request processing of the clones ends simultaneously. Due to the deterministic queueing discipline, the queue length and order of request clones are identical in each server at all times, implying that all request clones to every r^{orig} will begin and end their service at the same time.

---

and let $c \geq 0$ be the time between the completion of the first clone of r^{orig} and the cancellation $r_{1:n}^c$. Using these two assumptions, the following lemma can then be stated.

Lemma 6.1
All original requests in a cloning system under Assumptions 6.1 and 6.2 receive synchronized service, as long as the servers have the same k and deterministic queueing discipline.

Proof. Trivially, since for every r^{orig} its clones arrive and depart each server simultaneously at all times, and all servers experience the same deterministic queueing discipline, both the queue length and time-of-arrival order of the request clones must be the same in each server. Further, as the number of processors k are the same for each server, all request clones to each r^{orig} must start (and end) their service at the same time. □

A cloning system fulfilling Lemma 6.1 is said to be under synchronized service, and referred to as a *clone-to-all* system.

An illustrative image of a clone-to-all system can be seen in Figure 6.1. Deterministic queueing disciplines includes most commonly used ones, such as the introduced FCFS, PS and INF disciplines. Before continuing, we will for completeness state the following classic result regarding the CDF of the minimum over a set of stochastic variables.

Lemma 6.2
Given a set of n independent random variables $\{X_1, \ldots, X_n\}$, each with their distinctive CDF denoted as $\Phi_i(x)$ for X_i, the CDF of the random variable $X_{\min} = \min\{X_1, \ldots, X_n\}$ is given by

$$\Phi_{\min}(x) = 1 - \prod_{i=1}^{n} \{1 - \Phi_i(x)\}. \tag{6.1}$$

A similar, more complex result exists if $\{X_1, \ldots, X_n\}$ are dependent.

Proof. This is a well-known fact in statistics. It can be quickly shown by expressing $\Phi_{\min}(x) = \mathbb{P}(\min\{X_1, \ldots, X_n\} \leq x) = \mathbb{P}\left(\bigcup_{i \in 1..n} [X_i \leq x]\right)$ and expanding the union via the inclusion-exclusion principle [Modica and Poggiolini, 2012, Corollary 2.70]. If $\{X_1, \ldots, X_n\}$ are independent, the resulting formula reduces to (6.1). □

Now, if we assume that the arrivals and cancellations happen immediately, then the following theorem can be stated.

Theorem 6.1—The Equivalent G/G/K Model
Assume a clone-to-all system with $a = c = 0$, it will behave equivalently to a single G/G/k queue of the same discipline and k as the original servers with interarrival distribution Φ^{λ} and service time distribution $\Phi^{s}_{\min}$, determined according to Lemma 6.2 over $\Phi^{s}_{1:n}$.

Proof. Due to the assumptions necessary for synchronized service, the queue lengths and cloned request order are identical in each server at all times. The entire system can thus be equivalently thought of as a single k-processor queue of the same discipline, representing the queueing of all original requests. When r^{orig} arrives to the system, as $a = 0$ all its clones immediately arrives at the servers, hence Φ^{λ} is the interarrival time CDF to this new equivalent queue as well. Furthermore, the service of all clones of r^{orig} will start at the same time, and since $c = 0$ they will all end their service immediately at the completion of the first clone. Thus the service time of r^{orig} in this new equivalent queue becomes the minimum over the service times in all n servers. □

Remark 6.1
Theorem 6.1 does **not** require any assumptions on properties of either the interarrival distribution or the service time distributions. Furthermore, the theorem holds for any homogeneous (deterministic) queueing discipline and value of k across the servers.

The equivalent G/G/k model allows us to directly obtain utilization and throughput for the entire cloning system, and if there exists approximations

for retrieving, e.g., mean queue length for the G/G/k model, then these can be used to analyze the cloning system as well. If not, it is always possible to obtain the wanted metrics via simulation of the G/G/k model at a lower computational cost than simulating the entire cloning system.

In contrast to previous research that heavily relies on the FCFS discipline and specific properties of the interarrival and service time distributions, synchronized service and the equivalent G/G/k model extends the state of the art. However, we have essentially traded the assumptions on the involved queues for the assumptions necessary to obtain synchronized service. It is limiting that synchronized service requires a clone-to-all policy, and the assumptions of synchronized arrivals and cancellation are unrealistic to obtain in real settings. However, the clone-to-all system can be used as the basic building block for more complex structures where, for example, a load balancer can be placed in front of multiple such systems which is studied in the final part of Section 6.2. Furthermore, in Section 6.3 we study what will happen if we relax these two necessary assumptions under the PS discipline.

Demonstrating Equivalence via a Simulation Experiment

In this section, we demonstrate that the equivalent G/G/k model is in fact equivalent to a clone-to-all system, by performing a simulation experiment using the aforementioned cloning simulator.

Consider a cloning system of 3 servers under synchronized service. Each server is assumed to be modeled by a single PS queue with $k = 1$. Assume that the interarrival times are uniformly distributed between 0 and 4. Further, assume that the service time distributions are independent, but heterogeneous with the following distributions

(i) Exponential with $\Phi_1^s(t) = 1 - e^{-0.480t}$.

(ii) Weibull with $\Phi_2^s(t) = 1 - e^{-0.125t^3}$.

(iii) Uniform with $\Phi_3^s(t) = (t - 0.5)/3.5$ if $0.5 \leq t \leq 3.5$ else 0.

Using Theorem 6.1, we can model this clone-to-all system with an equivalent G/G/k model with the service time distribution $\Phi_{\min}^s(t) = 1 - \prod_{i=1}^{3} [1 - \Phi_i^s(t)]$. The interarrival time distribution of the equivalent G/G/k model is the same as for the original cloning system.

In total, 20 simulation experiments, each with 10^6 requests, where performed for both the cloning system and the equivalent model. In Figure 6.2 the the resulting empirical CDFs of the cloning system and the equivalent G/G/k model are shown to be practically identical.

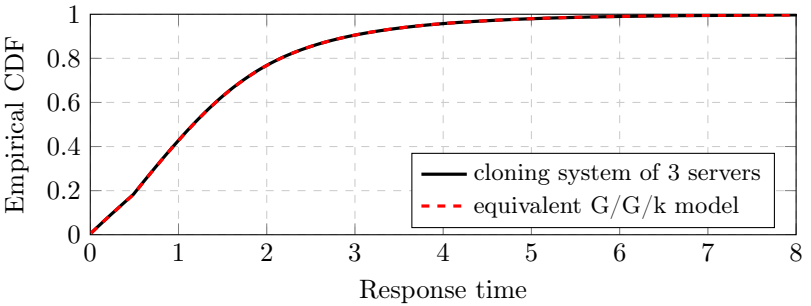


Figure 6.2 The empirical response time CDFs from a cloning system of 3 servers under synchronized service (black) and from the equivalent G/G/k queue (red dashed). The data was retrieved through 20 repeated simulations of 10^6 requests each, and the 95% confidence intervals lie within the lines.

6.2 Analyzing Synchronized Cloning Systems

In this section, we exemplify how the equivalent G/G/k model can be used to analyze cloning systems using existing results in queueing theory. This is done by investigating the optimal *cloning factor* $c_f \in \mathbb{Z}_+$, i.e., the number of clones for each request, with respect to the mean response time in systems under synchronized service. First, the baseline clone-to-all system will be considered. Later, we will show how a set of clone-to-all systems can serve as building blocks when modeling cloning systems behind a load balancer, and analyze the codesign between load balancer and cloning factor.

Although the equivalent G/G/k model is compliant with any interarrival and service time distributions we restrict ourselves to the PS discipline, $k = 1$ and Poisson arrivals with mean λ to simplify the analysis. To add a bit of realism and showcase how the G/G/k model can handle dependencies the S&Z model described next will be used for the service time distributions.

S&Z—A dependent service time model. The equivalent G/G/k model supports dependencies across service time distributions. In $\Phi^s_{\min}(t)$, these dependencies are normally represented by joint CDFs between service times, which is in general difficult to determine and analyze. In [Gardner et al., 2016a] a model is proposed that alleviates this issue by decoupling the task size of the original request r^{orig} from the server slowdowns affecting clones $r^c_{1:n}$. This is motivated by the fact that in real systems, the clones to an original request are all copies of the same task to be processed. Let Z^{orig} denote the original task size and $S^c_{1:n}$ the slowdowns, both are assumed to be independent stochastic variables. We will use the multiplicative version that expresses the service times $t^s_{1:n}$ of the clones $r^c_{1:n}$ as $t^s_{1:n} = Z^{orig} \cdot S^c_{1:n}$.

This service time model simplifies our analysis of dependent clones, as the server slowdowns $S_{1:n}$ can be viewed as *independent* across servers and original requests r^{orig}. Hence $t^s_{\min}$ can be written as the product distribution $t^s_{\min} = Z^{orig} \cdot S_{\min}$, where the CDF of $S_{\min}$ is determined according to (6.1). Calculating the distribution is cumbersome, but its first moment becomes

$$\mathbb{E}[t^s_{\min}] = \mathbb{E}[Z^{orig}] \cdot \mathbb{E}[S_{\min}]. \tag{6.2}$$

Hence, if the expected values for Z^{orig} and $S_{1:n}$ are known, the expected value can be obtained for the service time distribution of the equivalent G/G/k model. Similar results exists for higher moments.

Assumption 6.3
In the remainder of this section, we will model the server slowdowns $S_{1:n}$ using the empirical Dolly distribution, with the PMF defined in Table 6.1 for the Dolly(1,12) case. The task size Z^{orig} will be modeled using a two phase hyperexponential distribution with balanced means with $\mathbb{E}[Z^{orig}] = 1/4.7$ and squared coefficient of variation $\left(\mathbb{D}\left[Z^{orig}\right]/\mathbb{E}\left[Z^{orig}\right]\right)^2 = 2$. These particular values are chosen such that $\mathbb{E}\left[t^s_i\right] = \mathbb{E}\left[Z^{orig}\right] \cdot \mathbb{E}\left[S_i\right] = 1$ for any server i.

First published in [Ananthanarayanan et al., 2013], and later on used in e.g. [Gardner et al., 2016a], the Dolly distribution is a discrete distribution based on empirical data on server slowdowns from traces collected from Microsoft Bing's Dryad and Facebook's Hadoop clusters.

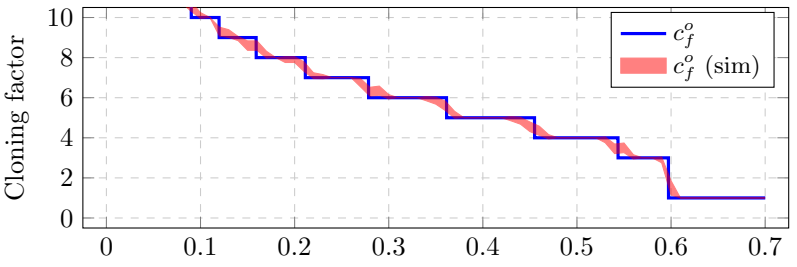
Finding Optimal Cloning Factors in Clone-to-All Systems

For the baseline clone-to-all system, the equivalent G/G/k model is directly obtainable. Whether evaluated by closed-form expression or simulation, it is possible to check all relevant cloning factors exhaustively to find the optimal versus some performance metric, as the cloning factor c_f is a one dimensional discrete variable. However, as a clone-to-all system implies that $n = c_f$, service times, arrival rates or processors needs to be scaled accordingly to make any meaningful comparisons.

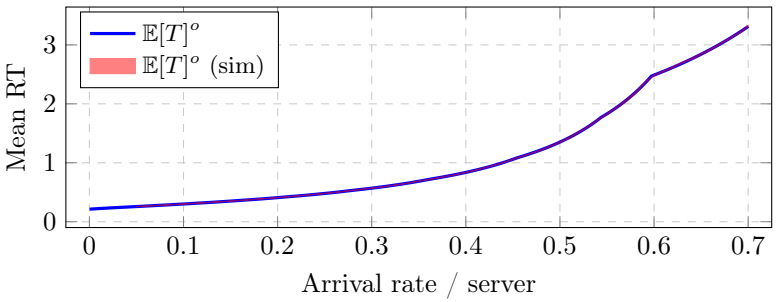
Regarding the mean response time, many closed-form expressions rely upon the first and second moments of the interarrival and service time distributions. As shown, these are in fact obtainable in a simple fashion from the equivalent G/G/k model, even for dependent service times using the S&Z model. Considering our example system with PS queues and $k = 1$ processors,

Table 6.1 The empirical Dolly(1,12) distribution from [Ananthanarayanan et al., 2013], used to model server slowdowns $S_{1:n}$.

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



(a) Optimal cloning factors.



(b) Corresponding mean response times.

Figure 6.3 Clone-to-all: (a) the optimal cloning factors together with (b) the corresponding optimal mean response times over an interval of arrival rates. The figure shows an comparison between theoretical values (blue line) and simulated 95% confidence interval of the means (red area).

and with Poisson arrivals and service times according to Assumption 6.3, the expression for $\mathbb{E}[T]$ becomes

$$\mathbb{E}[T] = \frac{\mathbb{E}[t^s_{\min}]}{1 - \lambda \mathbb{E}[t^s_{\min}]} = \frac{\mathbb{E}\left[Z^{orig}\right] \mathbb{E}[S_{\min}]}{1 - \lambda \mathbb{E}[Z^{orig}] \mathbb{E}[S_{\min}]}. \tag{6.3}$$

For stability, Equation (6.3) requires the utilization ρ of the clone-to-all system to be less than 1, or

$$\rho = \lambda \mathbb{E}\left[Z^{orig}\right] \mathbb{E}[S_{\min}] < 1. \tag{6.4}$$

This allows us to determine utilization, thus stability, and mean response time for our clone-to-all system for any cloning factor c_f. Using (6.3), we can thus find the optimal cloning factor c_f^o and the corresponding optimal mean response times $\mathbb{E}[T]^o$ using exhaustive search.

An example is shown in Figure 6.3. The top figure shows the optimal cloning factors c_f^o, whereas the bottom figure shows the corresponding optimal mean response times $\mathbb{E}[T]^o$. The blue lines show the theoretical values,

while the red areas show the 95% confidence interval of the means for the simulation results. As expected, the simulated c_f^o and $\mathbf{E}[T]^o$ follow their theoretical values closely. The comparison between cloning factors was performed such that the arrival rate per server is preserved, i.e., if a new server (and clone) is added λ is scaled accordingly. This could, for example, model joining multiple servers under similar arrival rates, to a single cloning system subjected to the combined arrivals. For each λ/server, the optimal cloning factor is found by checking each cloning factor $c_f \in 1:13$ and choosing the one that yields the smallest mean response time. All simulations where evaluated over 20 independent runs per combination of λ and c_f, each for 10^6 incoming requests.

As expected, higher cloning factors are more beneficial for lower system loads since the clones can utilize servers that otherwise would be mostly idle. For high system loads, the service time dependencies introduced in the S&Z model limit the use of cloning and for $\lambda > 0.6$ per server, no cloning ($c_f^o = 1$) is optimal.

Multiple Clone-to-All Systems Behind a Load Balancer

A partial relaxation of the clone-to-all assumption can be made by allowing cloning to subsets of servers, i.e., *clusters*, as proposed in [Joshi et al., 2017]. This is more natural, as it allows us to choose a smaller cloning factor c_f than the available servers n. The clone-to-all assumption can instead be fulfilled in each cluster, by cloning to all its servers for each incoming request to said cluster. A load balancer can then be placed in front of the clusters to determine which cluster to direct the clones of r^{orig}. We denote this a *clone-to-cluster* system, and define it as follows.

Definition 6.2—Clone-to-cluster
A cloning system of n serves partitioned into m clusters, each $j \in 1:m$ containing d_j servers s.t. $\sum_j d_j = n$. The servers in each cluster is assumed to have the same k and deterministic queueing discipline. The clusters are further assumed to lie behind some load balancer with policy ℓ, which directs incoming original requests r^{orig} to some cluster cluster j. The request is then cloned to all servers in cluster j according to Assumptions 6.1 and 6.2.

Each cluster in a clone-to-cluster system becomes its own clone-to-all system, and assuming $a = c = 0$, they each have an equivalently G/G/k model. Thus the entire clone-to-cluster system can be modeled as m G/G/k queues behind some load balancer. An illustration can be seen in Figure 6.4. In the rest of this chapter, we will for simplicity assume that $d_j = d \; \forall j \in 1:m$.

Analyzing a clone-to-cluster system is thus reliant on there existing expressions for the desired performance metrics of m G/G/k queues behind ℓ. This is true in certain cases, e.g., for random load balancing with Poisson

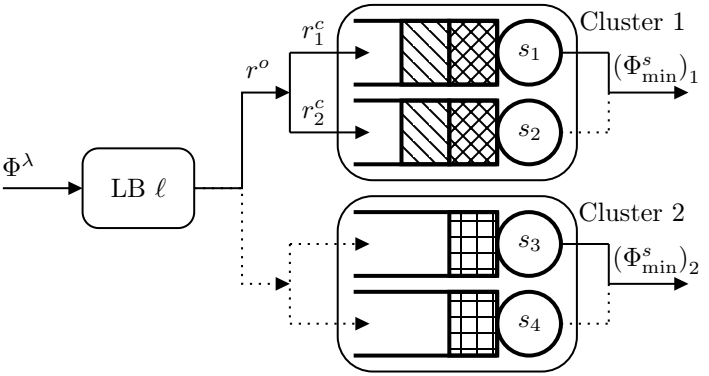


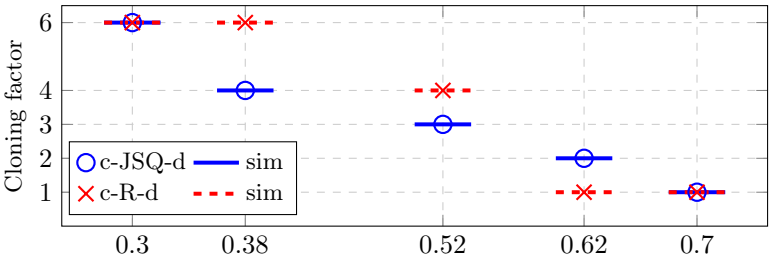
Figure 6.4 Clone-to-cluster: A cloning system where the $n = 4$ servers have been partitioned into $m = 2$ clusters of $d = 2$ servers each. The two clusters lie behind some load balancer with policy ℓ, which directs and clones the incoming requests to the clusters. Each cluster is assumed to fulfill the assumptions necessary to be considered a clone-to-all system on its own.

arrivals of rate λ, each G/G/k queue simply receives Poisson arrivals of rate λ/m. However, compared to the clone-to-all system it is in general harder to determine optimal cloning configurations. It boils down to choosing $\{d_j\}$ and m with respect to some performance metric and thus exhaustive search quickly becomes computationally infeasible for large n.

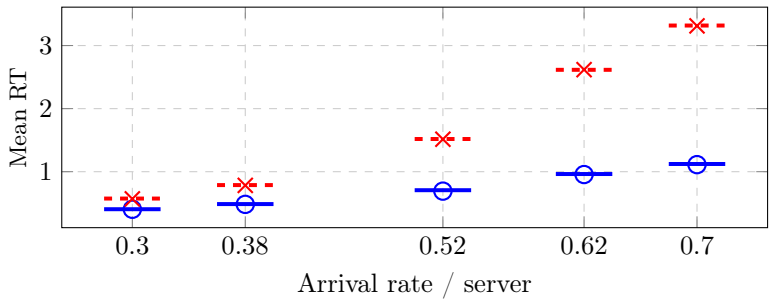
With our assumption $d_j = d \; \forall j \in 1 : m$, exhaustive search can however still be used. It implies that $m = {}^n/_d$, and thus finding the optimal configuration again becomes the search over a one dimensional discrete cloning factor $c_f = d$. However, the assumption restricts choices of d to systems where n and d are evenly divisible.

Codesign example using Random and JSQ. Assume $n = 12$ servers under the PS discipline with $k = 1$, which assuming equal cluster sizes implies that sizes $c_f = d \in \{1, 2, 3, 4, 6, 12\}$ are valid choices of cluster sizes. The extreme cases $c_f = 1$ corresponds to a system with no cloning, while $c_f = 12$ implies a clone-to-all system. As before, we assume Poisson arrivals and service times distributed according to the S&Z model defined in Assumption 6.3. The two load balancing policies considered are Random and JSQ.

Random: As the Random policy preserves Poisson arrivals to the clusters with rate ${}^\lambda/_m$, the exact analysis regarding the clone-to-all policy presented in the previous subsection is directly applicable to this codesign. We denote this codesign as cluster-Random-d (c-R-d).



(a) Optimal cloning factors.



(b) Corresponding mean response times.

Figure 6.5 Clone-to-cluster: (a) the optimal cloning factors, or cluster sizes, together with (b) the corresponding optimal mean response times over 5 different arrival rates. The figure shows an comparison between theoretical values for c-JSQ-d (blue circles) and c-R-d (red crosses), and simulated values for c-JSQ-d (blue) and c-R-d (red dashed).

JSQ: An expression of $\mathbb{E}[T]$ for a JSQ system with PS queues under Poisson arrivals can be obtained with the very accurate approximation presented in [Gupta et al., 2007a], given λ, m and the first moment of the service time distribution. As these are all obtainable for the equivalent G/G/k queue, the JSQ approximation can be used analyze this codesign which we denote as cluster-JSQ-d (c-JSQ-d).

In Figure 6.5, the theoretical and simulated optimal cloning factors (or equivalently cluster sizes) $c_f^o = d$ and corresponding optimal mean response times $\mathbb{E}[T]^o$ are shown. The blue circles and red crosses show the theoretical results for the c-JSQ-d and c-R-d strategies, respectively. The blue and red dashed lines show the respective simulated values, with the 95% confidence intervals within the displayed lines. As can be seen, the theoretical and simulated values match very well. The c-R-d match is exact as it is obtained using

exact analysis, while the simulated c-JSQ-d mean response times are slightly (1-3%) off as they are based on (highly accurate) approximations. Similar to the clone-to-all system example, we used exhaustive search to find the optimal cloning factor. All simulations where evaluated over 20 independent runs per combination of c_f, λ and ℓ, each for 10^6 incoming requests.

As can be seen, the c-JSQ-d cloning strategy is shown to give a lower mean response time than c-R-d for all possible arrival rates. This is expected, as the standard JSQ load balancer is well-known to outperform the random load balancer. As for the clone-to-all system, the trade-off between using idle servers and increasing the system load is clearly visible in the decreasing c_f^o over increasing λ. The fact that $c_f^o > 1$ for certain λ implies that in these load balancing scenarios, partitioning your servers into cloning clusters will actually lead to a reduction in the mean response time. However, for both codesigns, there exist arrival rates large enough for which the found $c_f^o = 1$, implying that no cloning is optimal in these cases.

6.3 Imperfect Synchronized Service

In this section we study the impact of relaxing the assumptions needed to obtain synchronized service. First, we consider the effects of imperfect arrivals and cancellations in the system. This is of high importance, as it is unrealistic to assume that it would be possible to design a perfectly synchronized service in practice. Second, we further relax the clone-to-all policy to allow for a more general cloning approach, for which we cannot guarantee synchronized service. Finally, the section is concluded with a larger simulation experiment to test the robustness of the equivalent G/G/k model to these relaxations. In order to analyze this imperfect synchronized service, we will limit ourselves to only considering server systems where the servers have $k = 1$ and follow the PS discipline. The following can be stated for such systems.

Remark 6.2
Given a cloning system of PS queues under synchronized service, all clones $r^c_{1:n}$ of an original request r^{orig} will experience *identical processor shares*, i.e., $p^s_i = p^s_j \ \forall i, j \in 1 : n$.

Introducing 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 canceling requests can occur. We model these imperfections in arrivals and cancellations using the notion of *arrival delays* and *cancellation delays*.

Definition 6.3—Arrival and cancellation delay
Let the arrival delay $a_i \geq 0$ be an SV representing the time difference between the original request arrival and cloned request arrival on s_i. Additionally, let the cancellation delay $c_i \geq 0$ be a SV representing the time difference between the first completed cloned request r_k^c on s_k and the cancellation of r_i^c.

It was shown in Lemma 6.1 that a system is in fact synchronized if for each original request $a_i = a_j$ and $c_i = c_j \; \forall i, j \in 1 : n$. Regarding cancellation delays, it have previously been studied for redundancy with equal delays in [Joshi, 2018] and with more general delays in [Lee et al., 2017], but these only considers scheduling policies for central queue systems.

Both types of delay will affect the system in different manners. Considering cancellation delays, the response time of a specific original request will not be directly affected, but clones not subject to immediate cancellation will linger and take up processing power from other requests. On the contrary, as arrival delays occur before the clones arrive at the servers, it will not interfere with the processing of other requests and instead only affect the response time directly. The presence of either delay type however ensures that the clone-to-all system is no longer synchronized, and thus the G/G/k model is no longer exact. However, it is possible to derive an explicit formula for an approximate upper bound on the expected response time $\mathbb{E}[T]$.

First, we assume that the distributions of a_i and c_i are independent and homogeneous, and that the service time distributions are homogeneous. Let t_i^s be a SV distributed according to $\Phi_i^s(t)$, and let T_i be the response time of clone r_i^c for some r^{orig}, i.e., the arrival delay a_i plus its time in the queue including the cancellation delay c_i. Assuming no cancellation delays (but potential arrival delays), $T_i = T_j \; \forall i, j \in 1 : n$ which is also the response time of r^{orig}. Let $v_i = {}^1/_{p_i^s} \geq 1$ be the inverse mean processor share in server s_i during the lifetime of r_i^c, and let $N = \sum_{i=1}^n v_i$, i.e., the total inverse processor share in the system during the lifetimes of the clones of an original request.

Disregarding the delays, conditioned on v the response time of r^{orig} to a single PS queue is given by vt^s. Thus, considering a cloning system following the PS discipline, the mean response time for some r^{orig} cloned to all n servers and conditioned on $\boldsymbol{v} \in \mathbb{R}_+^{n \times 1}$ can be expressed as

$$\mathbb{E}[T \mid \boldsymbol{v}] = \mathbb{E}[\min(\{v_j t_j^s\}_{j=1:n})]. \tag{6.5}$$

Assuming homogeneous service times, the following can be obtained.
Lemma 6.3
The expected response time of an original request cloned to n servers at a specific $N = \sum \boldsymbol{v}$ is maximized when all elements in $\boldsymbol{v}$ are equal, i.e.

$$\arg\max_{\boldsymbol{v}} \mathbb{E}[T \mid \boldsymbol{v}] = \boldsymbol{v}^u, \quad \text{where } v_i^u = \frac{N}{n} \; \forall i \in 1 : n. \tag{6.6}$$

Proof. (6.5) can be rewritten using the Law of Total Expectation

$$\sum_{k=1}^{n} \mathbb{E}[\min(\{v_j t_j^s\}_{j=1:n}) \mid t_k^s \leq \forall t_i^s] \cdot \mathbb{P}(t_k^s \leq \forall t_i^s), \tag{6.7}$$

as all t_i^s belong to the same distribution, $\mathbb{P}(t_k^s \leq \forall t_i^s) = 1/n$. Using that the minimum over a set is bounded by all of its members gives

$$\mathbb{E}[\min(\{v_j t_j^s\}_{j=1:n})] \leq \sum_{k=1}^{n} v_k \mathbb{E}[t_k^s \mid t_k^s \leq \forall t_i^s] \frac{1}{n}. \tag{6.8}$$

Again, as all t_i^s belong to the same distribution, we get that $\forall k\ \mathbb{E}[t_k^s \mid t_k^s \leq \forall t_i^s] = \mathbb{E}[\min(\{t_i^s\}_{i=1:n})]$ and thus

$$(6.8) = \frac{N}{n} \mathbb{E}[\min(\{t_j^s\}_{j=1:n})] = \mathbb{E}\left[\min\left(\left\{v_j^u t_j^s\right\}_{j=1:n}\right)\right]. \tag{6.9}$$

□

For a cloning system under synchronized service, we from Remark 6.2 have that $\boldsymbol{v} = \boldsymbol{v}^u$ at all times, but this is not the case for nonsynchronized service. This implies that, for any N, a system, when synchronized, will actually form an upper bound to the mean response time for the original request r^{orig}. To extrapolate the bound to system averages, we make the following statement.

Remark 6.3
Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two, possibly nonsynchronized due to delays, clone-to-all systems with the same number of servers n and arrival rate λ. If for all N, $\mathbb{E}[T_i \mid N, \mathcal{S}_1] \leq \mathbb{E}[T_i \mid N, \mathcal{S}_2]\ \forall i \in 1:n$, then intuitively it should hold that $\mathbb{E}[T \mid \mathcal{S}_1] \leq \mathbb{E}[T \mid \mathcal{S}_2]$.

This statement remains unproven. In our simulation campaigns we have found no counterexamples, but corner cases for specific choices of delays and distributions might exist.

It is now possible to compute bounds, although approximate, on the effects of arrival and cancellation delays on the expected response time, by letting $\mathcal{S}_1$ be a system affected by delays and $\mathcal{S}_2$ a constructed synchronized system, such that $\mathcal{S}_1$ and $\mathcal{S}_2$ fulfill the conditions in Remark 6.3. Since $\mathcal{S}_2$ is synchronized, the equivalent G/G/k model can be applied directly to explicitly compute the bounds for $\mathcal{S}_1$. We proceed by first considering the two delays separately.

Proposition 6.1—Upper bound considering arrival delays
Let $\mathcal{S}_1$ be a clone-to-all system with arrival delay, and $\mathcal{S}_2$ an identical system with no delay. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be subjected to the same arrival rate. Then

$$\mathbb{E}[T \mid \mathcal{S}_1] \lessapprox \mathbb{E}[T \mid \mathcal{S}_2] + \mathbb{E}[a]. \tag{6.10}$$

Proof. Consider $\mathcal{S}_1$. For a specific $N = \sum \boldsymbol{v}$, due to the absence of cancellation delays, all clones will have the same response time T given by

$$T \mid \boldsymbol{v}, \mathcal{S}_1 = \min\{a_j + v_j t_j^s\}_{j=1:n}, \tag{6.11}$$

Following Lemma 6.3, the expected response time can be bounded as

$$\begin{aligned}\mathbb{E}[T \mid \boldsymbol{v}, \mathcal{S}_1] &\leq \sum_{k=1}^{n} \mathbb{E}[a_k + v_k t_k^s \mid t_k^s \leq \forall t_i^s] \frac{1}{n} \\ &= \mathbb{E}[a] + \mathbb{E}[\min\{v_j^u t_j^s\}_{j=1:n}]. \end{aligned} \tag{6.12}$$

This is equivalent to system $\mathcal{S}_1$, but where the individual stochastic arrival delays have been replaced by the constant arrival delay $\mathbb{E}[a]$. By Remark 6.3, $\mathbb{E}[T \mid \mathcal{S}_1]$ is approximately bounded by its expected response time. Further, a clone-to-all system with a constant arrival delay is synchronized according to Lemma 6.1. Thus, the expected response time can be expressed as $\mathbb{E}[T \mid \mathcal{S}_2]$ which can be obtained using the equivalent G/G/k model, plus the additive delay. □

Note that for Proposition 6.1, $\mathbb{E}[a]$ does not affect the stability of $\mathcal{S}_2$ as clones during their arrival phase do not affect the processing speed. Thus, if $\mathcal{S}_2$ is stable, then so should $\mathcal{S}_1$ regardless of arrival delays.

Proposition 6.2—Upper bound considering cancellation delays Let $\mathcal{S}_1$ be a clone-to-all system with cancellation delays and $\mathcal{S}_2$ be an identical system without delay and with service time $t_i^s \mid \mathcal{S}_2 = (t_i^s \mid \mathcal{S}_1) + \mathbb{E}[c]$. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be subject to the same arrival rate. Then

$$\mathbb{E}[T \mid \mathcal{S}_1] \lessapprox \mathbb{E}[T \mid \mathcal{S}_2]. \tag{6.13}$$

Proof. Consider $\mathcal{S}_1$. For a specific $N = \sum \boldsymbol{v}$, the response time of each clone to an original request becomes

$$T_i \mid \boldsymbol{v}, \mathcal{S}_1 = \min\{v_j t_j^s\}_{j=1:n} + \min\left(c_i, v_i t_i^s - \min\{v_j t_j^s\}_{j=1:n}\right). \tag{6.14}$$

The second minimum incorporates the possibility that a cloned request completes after $\min\{q_j t_j^s\}_{j=1:n}$ but before c_i. Thus, the expected response time of each clone can be bounded by

$$\mathbb{E}[T_i \mid \boldsymbol{v}, \mathcal{S}_1] \leq \mathbb{E}\left[\min\{v_j t_j^s\}_{j=1:n}\right] + \mathbb{E}[c_i]. \tag{6.15}$$

Contrary to arrival delays, the cancellation delays will have a direct impact on the processing speed of the servers. Hence, it cannot be isolated in the

same manner as the arrival delays. Instead, it can be incorporated into the service times as follows.

$$\begin{aligned}\mathbb{E}\left[T_i \mid \boldsymbol{v}, \mathcal{S}_1\right] &\leq \mathbb{E}\left[\min\{v_j^u t_j^s\}_{j=1:n}\right] + \mathbb{E}\left[c\right] \\ &\leq \mathbb{E}\left[\min\{v_j^u \left(t_j^s + \mathbb{E}\left[c\right]\right)\}_{j=1:n}\right] \\ &= \mathbb{E}[T_i \mid \boldsymbol{v}^u, \mathcal{S}_2], \end{aligned} \tag{6.16}$$

using Lemma 6.3 in the first step, and the fact that by definition $v \geq 1$ in the second step. This is equivalent to system $\mathcal{S}_1$ but where the service rates have been increased with the expected cancellation delay, i.e., $\mathcal{S}_2$, and it has synchronized service as it is subjected to no delay. Remark 6.3 then yields the original statement, and the expected response time of $\mathcal{S}_2$ can be obtained via the equivalent G/G/k model. □

Note that for Proposition 6.2, for large $\mathbb{E}[c]$ the upper bound can become infinite despite the potential stability of $\mathcal{S}_1$. Thus, the arrival rate of the system has to be less than $1/(\mathbb{E}[t^s] + \mathbb{E}[c])$ for both $\mathcal{S}_1$ and $\mathcal{S}_2$ to be stable. In addition, in the second step of (6.16) we need to multiply $\mathbb{E}\left[c\right]$ by the factor $N/n \geq 1$, potentially leading to large errors at higher utilization.

The following proposition shows that the effect of both arrival and cancellation delays is additive.

Proposition 6.3—Upper bound considering combined delays
Let $\mathcal{S}_1$ be a clone-to-all system with arrival and cancellation delays, and $\mathcal{S}_2$ an identical system but without delays and $t_i^s \mid \mathcal{S}_2 = (t_i^s \mid \mathcal{S}_1) + \mathbb{E}[c]$. Let both systems be subject to the same arrival rate. Then,

$$\mathbb{E}[T \mid \mathcal{S}_1] \lessapprox \mathbb{E}[a] + \mathbb{E}[T \mid \mathcal{S}_2]. \tag{6.17}$$

Proof. Consider $\mathcal{S}_1$. For a specific $N = \sum \boldsymbol{v}$, the response time for each clone to an original request becomes

$$\begin{aligned} T_i \mid \boldsymbol{v}, \mathcal{S}_1 = &\min\{a_j + v_j t_j^s\}_{j=1:n} \\ &+ \min\left(c_i, a_i + v_i t_i^s - \min\{a_j + v_j t_j^s\}_{j=1:n}\right). \end{aligned} \tag{6.18}$$

Following the proofs of Propositions 6.1 and 6.2, the expected response time can be bounded as follows.

$$\begin{aligned}\mathbb{E}\left[T_i \mid \boldsymbol{v}, \mathcal{S}_1\right] &\leq \mathbb{E}\left[a\right] + \mathbb{E}\left[\min\{v_j^u t_j^s\}_{j=1:n}\right] + \mathbb{E}\left[c\right] \\ &\leq \mathbb{E}\left[a\right] + \mathbb{E}\left[\min\left\{v_j^u \left(t_j^s + \mathbb{E}\left[c\right]\right)\right\}_{j=1:n}\right]. \end{aligned} \tag{6.19}$$

This is equivalent to system $\mathcal{S}_1$, but with a constant arrival delay, no cancellation delay, and where service times have been increased with $\mathbb{E}\left[c\right]$. Hence,

following the proofs of Propositions 6.1 and 6.2, this system has synchronized service and its expected response time can be obtained using the equivalent G/G/k model on $\mathcal{S}_2$, plus the additive delay. □

The benefit of Propositions 6.1-6.3 is twofold. First, they show that small imperfections are not detrimental when trying to implement synchronized service in practice. Furthermore, as long as the expected response time of the equivalent G/G/k model is computable, these approximate bounds will also be computable. However, the results are only valid if one assumes that $a_i,\ c_i,\ t_i^s$ are homogeneous and known, which is not the case for all systems.

Fully Relaxing the Clone-to-All Assumption

The partial relaxation of the clone-to-all assumption introduced in the end of Section 6.2, by clustering the servers to a clone-to-cluster system, shows that it is indeed possible to obtain a cloning system under synchronized service when $c_f < n$. However, pre-partitioning the servers into clusters is superfluous. In practice, a more natural approach would instead be to allow the load balancing strategy ℓ to, for each original request, freely choose $c_f = d$ unique servers from $s_{1:n}$ to clone to. We denote this a *clone-to-any* system and define it as follows.

Definition 6.4—Clone-to-any
A cloning system of n serves behind some load balancer with policy ℓ, which takes incoming original requests r^{orig}, chooses d unique servers, and clones the request to these servers.

Let a-ℓ-d denote a clone-to-any system with load balancing policy ℓ. Given a Random load balancing policy, a-R-d has obvious similarities to the Redundancy-d system [Gardner et al., 2016b].

Clone-to-any can be seen as a cloning system with a full relaxation to the clone-to-all assumption, and thus it does not experience synchronized service. This implies that the equivalent G/G/k model is not directly applicable. However, a clone-to-cluster system c-ℓ-d with the same load balancing policy ℓ and cloning factor $c_f = d$ as some clone-to-all system a-ℓ-d could potentially be used as an approximation for obtaining desired performance metrics. To reason about such an approximation, we introduce the notion of *synchronization error* ϵ to quantify the imperfectness of the synchronization. Regarding the PS discipline, it is defined as follows.

Definition 6.5—Synchronization error in PS queues
The synchronization error ϵ in a cloning system of PS queues is defined as the coefficient of variation of the processor shares for the clones $r^c_{1:d}$ of an original request, i.e., $\epsilon = \mathbb{D}[p^s_{1:d}]/\mathbb{E}[p^s_{1:d}]$.

For $\mathbb{E}[\epsilon] > 0$, the system is nonsynchronized, for $\mathbb{E}[\epsilon] = 0$ it is synchronized and for small $\mathbb{E}[\epsilon]$ the system is *near-synchronized*. An a-ℓ-d system that experiences near-synchronized should perform similarly to its c-ℓ-d counterpart. Furthermore,

Remark 6.4
Consider a clone-to-any system, then $\rho \to 0$ implies that $\mathbb{E}[\epsilon] \to 0$.

The lower the utilization ρ, the higher the probability becomes that all clones $r^c_{1:d}$ to an original request r^{orig} execute alone on their servers. In the limit this implies that $p^c_i \to 1$ for all r^c_i and $\mathbb{E}[\epsilon] = 0$.

Thus, using c-ℓ-d it should be possible to derive accurate approximations of a-ℓ-d under low utilizations, for any ℓ and $c_f = d$. For more general 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 of the same original request, a-ℓ-d will be near-synchronized.

Evaluating Relaxations via Simulation Experiments

To perform a general evaluation of what happens to the performance of systems when the assumptions regarding synchronized service are relaxed as discussed in this section, we conducted two randomized simulation experiments. In the first simulation experiment, we analyzed the impact of arrival and cancelation delays to the performance of a clone-to-all system and the accuracy of the derived upper bound. In the second experiment, we analyzed the synchronization error for two clone-to-any systems using the load balancing strategies random and JSQ, denoted a-R-d and a-JSQ-d, and the subsequent $\mathbb{E}[T]$ approximation using the corresponding clone-to-cluster strategy.

The randomized simulation experiments were performed over 1000 random scenarios, each with 10^6 incoming requests from Poisson arrivals. For each scenario, we randomly selected a service time distribution from the following list.

(i) S&Z model defined in Assumption 6.3.

(ii) Exponential distribution with $\mu = 1$.

(iii) Weibull distribution with $\Phi^s(t) = 1 - \exp\left[-\left(\frac{t}{0.5}\right)^{0.5}\right]$.

(iv) Pareto distribution with $\Phi^s(t) = 1 - \left(\frac{0.6}{t}\right)^{2.5}$ if $t \geq 0.6$ else $\Phi^s(t) = 0$.

(v) Uniform distribution with $t^s_i \in [0, 2]$.

The particular shape and scale values were chosen such that the mean service time of all the above distributions at cloning factor $c_f = 1$ is $\mathbb{E}[t^s] = 1$. Furthermore, we randomly selected a number of servers n^{sim} and a cloning

factor c_f^{sim}, whose sets are defined separately in the two simulation experiments shown below. Finally, we randomly selected a utilization from the set $\rho^{sim} \in (1:9)/10$ and fit the arrival rate λ accordingly.

Arrival and cancellation delays. In this experiment, we considered the following cloning factor/servers $c_f^{sim} = n^{sim}$ and normalized delay τ^{sim}.

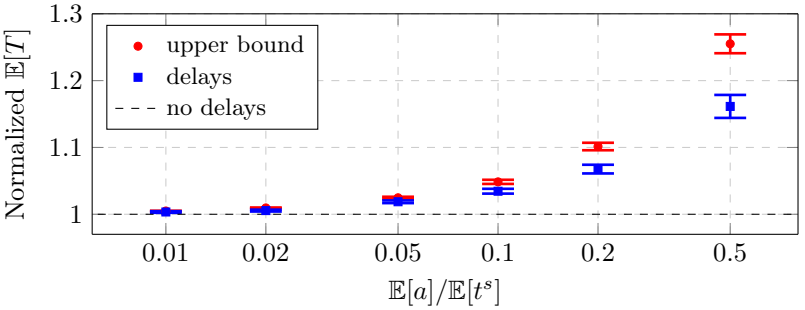
$$\begin{aligned} c_f^{sim} &= s_n^{sim} \in 2:10, \\ \tau^{sim} &\in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.5\} \cdot \mathbb{E}[t^s]. \end{aligned} \tag{6.20}$$

For these parameter sets, we reran the randomized simulation experiment three times considering (a) only arrival delays with $\mathbb{E}[a] = \tau^{sim}$, (b) only cancellation delays with $\mathbb{E}[c] = \tau^{sim}$, and (c) both delays with $\mathbb{E}[a] = \gamma\tau^{sim}$ and $\mathbb{E}[c] = (1-\gamma)\tau^{sim}$ where $0 < \gamma < 1$ was uniformly distributed. Both arrival and cancelation delays were assumed to be exponentially distributed.

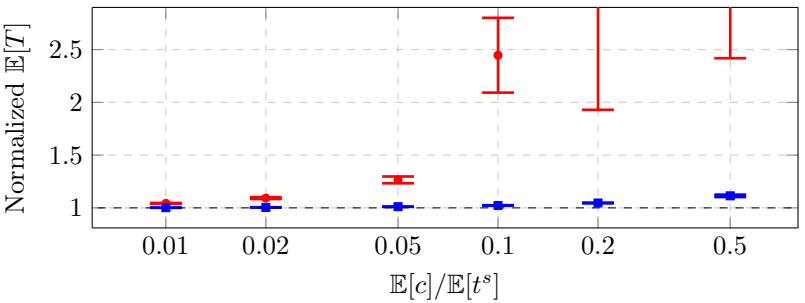
The simulation results together with the approximate bounds can be seen in Figure 6.6. Here, we have normalized $\mathbb{E}[T]$ from each random scenario to the corresponding theoretical value when no delays are present. The blue span shows the simulated $\mathbb{E}[T]$ when the delays are present, while the red span shows the upper bound. The interval in each span represents the 95% confidence interval of the corresponding random scenarios.

Here it can be seen that for low normalized delays (0.01-0.05), all normalized $\mathbb{E}[T]$ are close to 1. For higher delays, the normalized $\mathbb{E}[T]$ increases slightly to a maximum of 15-20% relative error compared to the synchronized clone-to-all without delays in all three cases. At a first glance this might look strangely small given the fact that the maximum mean delays are 50% of the service times. But remember that the arrival delays affects the response time directly, but are encompassed by the minimum as well. Additionally, cancellation delays affect only the response time indirectly by slowing down the system with the remaining clones. There will be servers where the clones to a completed r^{orig} depart quickly, and these make room for other clones to complete their processing. Also, at long cancellation delays, remaining clones will have a significant probability of actually completing before its cancellation time. Finally, the tight spans indicate that the delay induced relative error in clone-to-all under PS is robust to different utilizations, service time distributions, and cloning factors. This is not as surprising, as it is well-known that $\mathbb{E}[T]$ for M/G/1-PS queues are robust to the choice of G.

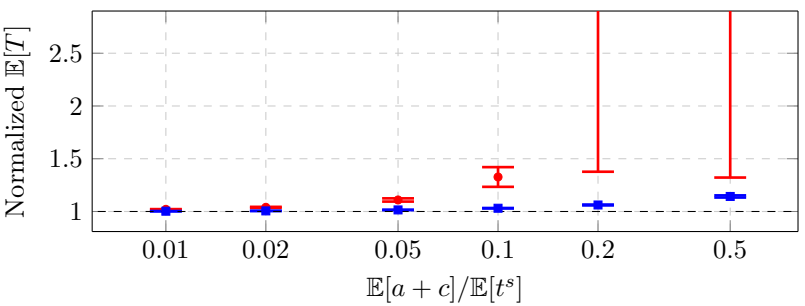
Hence, exponentially distributed delays whose mean are within 5% of the mean service time will have no large effect on clone-to-all systems. For larger mean delays, the impact will be noticeable but not very large. Thus, the equivalent G/G/k queue can be used to approximately model clone-to-all systems with high accuracy, even when rather large arrival and cancellation delays are present.



(a) Considering only arrival delays.



(b) Considering only cancellation delays.



(c) Considering both types of delays.

Figure 6.6 Arrival and cancellation delay simulation experiment. Compares the normalized $\mathbb{E}[T]$ from simulation (blue span) to the upper bound (red span) over six different normalized mean delays. The normalization for $\mathbb{E}[T]$ is performed such that each value is divided by the corresponding theoretical value without delays. The intervals represent 95% confidence intervals of all associated random scenarios. The legend in (a) applies to all figures. Note that the values on the x-axis has a logarithmic scaling.

Considering the upper bounds, for the low normalized delays (0.01-0.05) they are tight in all three experiments. However, as expected, for the larger delays the bounds for the cancellation delay become very large, and for some scenarios with mean delay 0.2 and 0.5 they even become infinite, as the upper bound cannot guarantee stability for these cases. As none of our simulated scenarios was unstable, it is obvious that the cancellation delay bound has limited usage for these higher delays. On the contrary, the bounds for the arrival delays are tight even at large delays.

Clone-to-any systems. In the second experiment, the synchronization errors of clone-to-any systems a-ℓ-d were studied and the resulting mean response times compared to the corresponding clone-to-cluster c-ℓ-d system. Here, we let

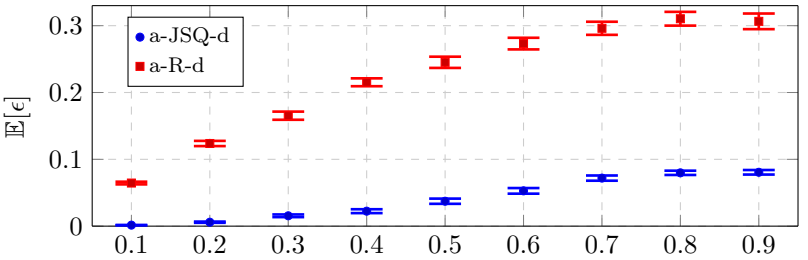
$$n^{sim} \in \{4, 6, 9, 12, 15, 21, 27, 30, 45, 48\}\,. \tag{6.21}$$

In each experiment, we randomly choose a n^{sim} and subsequently an evenly divisible c_f. This enables us to use a c-ℓ-d system with the same cloning factor $c_f = d$ as and load balancing policy ℓ as a-ℓ-d to form an approximation of its mean response time $\mathbb{E}[T]$.

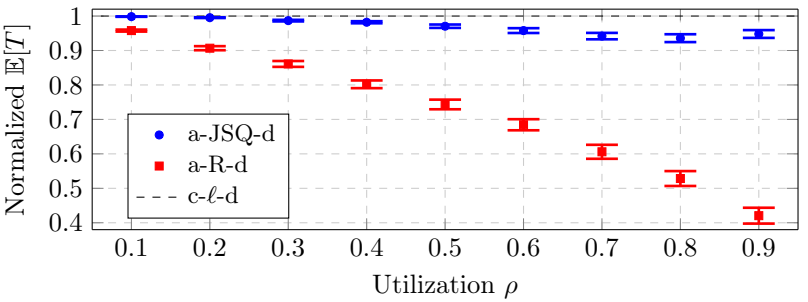
The results of the experiment can be seen in Figure 6.7 considering a-ℓ-d for both JSQ (blue span) and Random (red span) load balancing. The interval in each span represents the 95% confidence interval of the corresponding random scenarios. The upper plot shows the mean synchronization errors $\mathbb{E}[\epsilon]$, while the lower plot shows the normalized $\mathbb{E}[T]$. This normalization is performed by dividing each $\mathbb{E}[T]$ from a-ℓ-d obtained by simulation, to the theoretical value from its c-ℓ-d counterpart.

It can be clearly seen that a-JSQ-d gives a much lower synchronization error than a-R-d. This implies that a-JSQ-d is a much better approximation of its synchronized clone-to-cluster counterpart than a-R-d. This is further confirmed by considering the normalized $\mathbb{E}[T]$, where the values for a-JSQ-d are much closer to 1. For low values of the utilization ρ, both codesigns approximate the synchronized behavior fairly well as discussed in Remark 6.4. Further, the fact that a-JSQ-d has a small synchronization error and $\mathbb{E}[T]$ error for all utilizations suggests that a-JSQ-d experiences a near-synchronized service regardless of the arrival rate. This can be intuitively be explained by considering the JSQ algorithm. As JSQ always sends each batch of clones to the servers with the least amount of running requests, it will over time cause the servers to have a similar number of running requests present at all times. The clones $r^c_{1:d}$ of the same original request r^{orig} will then receive similar processor shares, leading to small synchronization errors.

Further, just as for the arrival and cancelation delay experiments, the tight spans indicate a robustness to the choice of servers, cloning factor, utilization, and service time distributions. As this experiment is also conducted over M/G/1-PS queues, this is not that surprising.



(a) Mean synchronization errors.



(b) Corresponding normalized mean response times.

Figure 6.7 Clone-to-all: Compares (a) mean synchronization error $\mathbb{E}[\epsilon]$ for a-JSQ-d (blue span) and a-R-d (red span), and (b) the corresponding normalized $\mathbb{E}[T]$ to their c-ℓ-d counterparts, over increasing utilization. The normalization of $\mathbb{E}[T]$ is performed such that each value is divided by the value for the c-ℓ-d counterpart. The intervals represent 95% confidence intervals for all associated random scenarios.

Looking more closely at the normalized $\mathbb{E}[T]$, 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 Lemma 6.3, but no proof as so far been found.

6.4 Extension to Speculative Execution

In this section, we develop a novel approximation method that allows us to analyze clone-to-any systems that has been further relaxed to allow for speculative execution. The approximation can be seen as an extension to us-

ing clone-to-cluster to approximate clone-to-any systems, and can apart from speculative execution also be used in these cases when d and n are not evenly divisible. It builds on assuming synchronized service to model the system using a set of equivalent G/G/k queues behind a load balancer, but where the increased system load from the influx of (speculative) clones is modeled as a reduction in the available servers. The synchronized service assumptions will not hold for the general case, but implies that the new approximation method will perform better with smaller synchronization errors ϵ.

We assume that our system of n servers lies behind some load balancer with strategy ℓ. Each server is assumed to have $k = 1$ and follow the PS discipline with service rate μ. Furthermore, requests arrive to the load balancer with rate λ. No specific distributions are assumed; however, for simplicity, we require the service time distributions to be homogeneous and independent across all n servers.

In speculative execution, instead of sending d clones at every arrival of an original request r^{orig}, we first dispatch r^{orig} to the servers via ℓ. After some specified amount of service time has passed for r^{orig}, we dispatch a speculative clone r_i^s to a unique server in the system via ℓ. Define δ_i as the completed service time when r_i^s is dispatched to the server system and $\mathcal{D}_d = \{\delta_1, \delta_2, \ldots, \delta_d\}$ as the ordered set of the service times of all speculative clones with $\delta_{i-1} \leq \delta_i \ \forall i \in 1 : d$. We assume immediate arrivals and perfect cancellations. Therefore, r_i^s arrives at a server at δ_i and when the original request or any of the dispatched speculative clones is completed, the rest are immediately cancelled.

New Service Time Distribution

Until the first speculative clone is dispatched at $t = \delta_1$, the original request r^{orig} will run without any clones in the system and thus its service time for $t \leq \delta_1$ simply becomes $\Phi^s(t)$.

When the first speculative clone r_1^s is dispatched, the immediate arrivals and PS discipline implies that r_1^s enters service immediately at δ_1. All speculative clones will have the same service time distribution $\Phi^s(t)$ as the original request, but conditioned on the fact that it enters service at $t = \delta_i$ for r_i^s. Hence, if we assume synchronized service and given that the service has not completed at $t = \delta_1$, the new service time for $t > \delta_1$ becomes the minimum between the service time of r^{orig} conditioned on $t > \delta_1$ and r_1^s conditioned on service start at δ_1. Following Lemma 6.2, the corresponding CDF of the minimum thus becomes

$$\Phi_1^c(t) = 1 - \left[1 - \Phi^s\left(t \mid \delta_1 < t\right)\right] \cdot \left[1 - \Phi^s\left(t - \delta_1\right)\right] \qquad \delta_1 < t. \tag{6.22}$$

To create the service time distribution of the original request and its first speculative clone, the probability that r^{orig} completes service before $t = \delta_1$,

i.e., $\Phi^s(\delta_1)$ needs to be taken into account. This can be done by weighting Φ_1^c at the breakpoint δ_1 with the remaining probability $1 - \Phi^s(\delta_1)$ which yields the following service time distribution

$$\Phi_1^s = \begin{cases} \Phi^s(t) & t \leq \delta_1, \\ \Phi^s(\delta_1) + [1 - \Phi^s(\delta_1)] \cdot \Phi_1^c(t) & \delta_1 < t. \end{cases} \tag{6.23}$$

Furthermore, when the second speculative clone is dispatched, the original request, together with its first speculative clone, has been running for $t = \delta_2$ and $t = \delta_2 - \delta_1$ time units, respectively. In the same manner as for r_1^s, given that the service time has not completed before $t = \delta_2$, the new service time for $t > \delta_2$ becomes the minimum between the service times of (i) r^{orig} conditioned on $t > \delta_2$, (ii) r_1^s conditioned on both service start at δ_1 and $t > \delta_2$, and (iii) r_2^s conditioned on service start at δ_2. As we have an expression for the minimum CDF of r^{orig} and r_1^s, following Lemma 6.2 the new minimum CDF becomes

$$\Phi_2^c(t) = 1 - [1 - \Phi_1^s(t \mid \delta_2 < t)] \cdot [1 - \Phi^s(t - \delta_2)] \qquad \delta_2 < t, \tag{6.24}$$

and the service time CDF can be created by taking into account the probability that the original request and the first speculative clone completes before $t = \delta_2$, i.e., $\Phi_1^s(\delta_2)$. This leads to the following iterative formula for determining the service time CDF Φ_d^s for the speculation scenario $\mathcal{D}_d$:

$$\Phi_d^s(t) = \begin{cases} \Phi_0^s(t) = \Phi^s(t) & t \leq \delta_1, \\ \quad \vdots & \\ \Phi_{i-1}^s(\delta_i) + \left(1 - \Phi_{i-1}^s(\delta_i)\right) \cdot \Phi_i^c(t) & \delta_{i-1} < t \leq \delta_i, \\ \quad \vdots & \\ \Phi_{d-1}^s(\delta_d) + \left(1 - \Phi_{d-1}^s(\delta_d)\right) \cdot \Phi_d^c(t) & \delta_d < t, \end{cases} \tag{6.25}$$

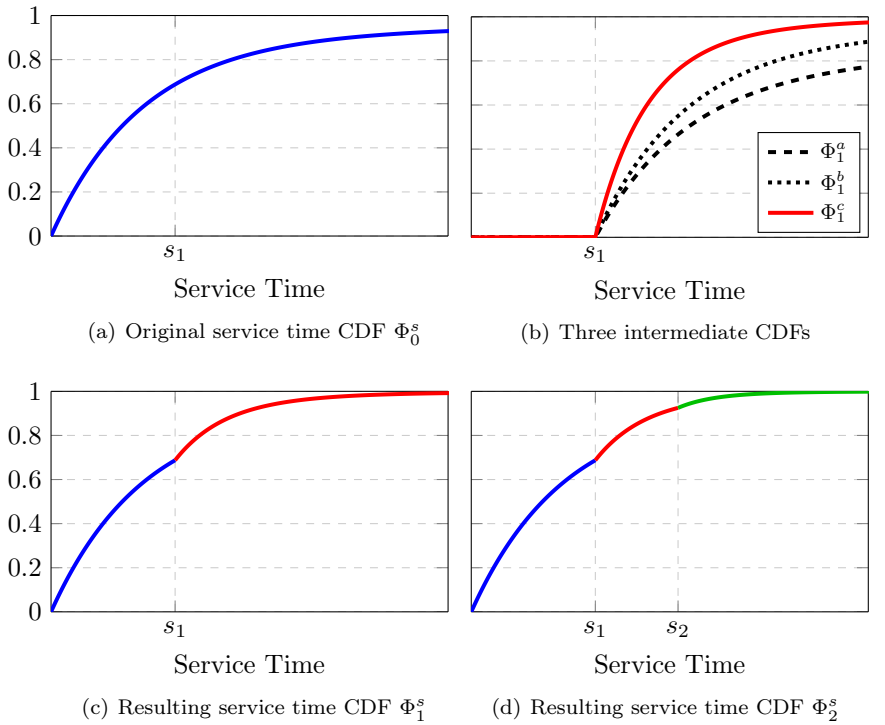
with the intermediate CDF $\Phi_i^c(x)$ determined as

$$\begin{aligned} \Phi_i^a(t) &= \Phi_{i-1}^s(t \mid \delta_i < t), \\ \Phi_i^b(t) &= \Phi^s(t - \delta_i), \\ \Phi_i^c(t) &= 1 - [1 - \Phi_i^a(t)] \cdot \left[1 - \Phi_i^b(t)\right]. \end{aligned} \tag{6.26}$$

The algorithm is visualized in Figure 6.8 for an example scenario $\mathcal{D}_2$.

Approximating System Load and Response Times

To obtain approximations on the response time, just as for the clone-to-cluster strategy, an appropriate method is needed to approximate $\mathbb{E}[T]$ for



(a) Original service time CDF Φ_0^s

(b) Three intermediate CDFs

(c) Resulting service time CDF Φ_1^s

(d) Resulting service time CDF Φ_2^s

Figure 6.8 Speculative cloning for an example scenario $\mathcal{D}_2 = \{\delta_1, \delta_2\}$. From Φ_0^s at δ_1, the three intermediate CDFs are formed using equations (2)-(4). Then Φ_1^c is added to Φ_0^s at δ_1 according to (6.25) to form Φ_1^s. The procedure is then repeated at δ_2 to form Φ_2^s.

the load balancing strategy ℓ over G/G/k queues. For example, the aforementioned methods for random and JSQ load balancing [Gupta et al., 2007a] considering M/G/1-PS queues require values on the arrival rate λ, the service rate μ and the available servers n, which we will here derive for an arbitrary speculative execution scenario $\mathcal{D}_d$. As synchronized service is assumed to derive the joint service time distribution, the accuracy of the approximation depends on how near-synchronized the resulting system under ℓ becomes.

From (6.25), we can calculate the new service rate $\mu(\mathcal{D}_d)$ for a scenario using d speculative clones as

$$\mu(\mathcal{D}_d) = \left(\int_0^\infty [1 - \Phi_d^s(t)]\, dt \right)^{-1}. \tag{6.27}$$

We define the *service factor* $f^{\mu}(\mathcal{D}_d)$ as the normalized increase of $\mu(\mathcal{D}_d)$ compared to the original $\mu(\mathcal{D}_0) = \mu$:

$$f^{\mu}(\mathcal{D}_d) = \frac{\mu(\mathcal{D}_d)}{\mu}. \tag{6.28}$$

To model changes in system load, we need to consider the amount of speculative clones sent for each original request r^{orig} and the time they spend in the system. We define the *speculation factor* f_i^p for a speculative clone at time δ_i as the probability $f_i^p = 1 - \Phi_i^s(\delta_i)$ that the clone is sent. Furthermore, we define the *sojourn factor* f_i^s for a speculative clone sent at time δ_i as its time spent in the system compared to the original request r^{orig}

$$f_i^s = \frac{\int_{\delta_i}^{\infty} \left[1 - \Phi_d^s(t \mid \delta_i < t)\right] dt}{\int_0^{\infty} \left[1 - \Phi_d^s(t)\right] dt}. \tag{6.29}$$

Now, for each speculative clone we have a probability of it being sent, and the time it spends in the system compared to the original request. We can thus define the *arrival factor* $f^{\lambda}(\mathcal{D}_d)$ for the total contributions to system load from all d speculative clones as

$$f^{\lambda}(\mathcal{D}_d) = 1 + \sum_{i=1}^{d} f_i^p \cdot f_i^s. \tag{6.30}$$

Finally, the *load factor* $f^{\rho}(\mathcal{D}_d)$ can then be defined as

$$f^{\rho}(\mathcal{D}_d) = \frac{f^{\lambda}(\mathcal{D}_d)}{f^{\mu}(\mathcal{D}_d)}. \tag{6.31}$$

If $f^{\rho}(\mathcal{D}_d) > 1$, speculative cloning in scenario $\mathcal{D}_d$ results in an increase of the original system load ρ, whereas $f^{\rho}(\mathcal{D}_d) < 1$ represents a decrease. Furthermore, we can define the modeled utilization of scenario $\mathcal{D}_d$ as

$$\rho(\mathcal{D}_d) = f^{\rho}(\mathcal{D}_d) \cdot \rho. \tag{6.32}$$

Equation (6.32) is very useful as it allows us to reason about the stability for the scenario $\mathcal{D}_d$. Note that the arrival factor $f^{\lambda}(\mathcal{D}_d) \geq 1$ does not imply an increase in the arrival rate of original requests r^{orig}. 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 n as

$$n(\mathcal{D}_d) = \frac{n}{f^{\lambda}(\mathcal{D}_d)}. \tag{6.33}$$

As a result, $n(\mathcal{D}_d)$ is defined as a positive real number, and not an integer. This might seem strange, as it has no direct connection to real systems.

However, as long as the scheme to obtain $\mathbb{E}[T]$ for load balancing strategy ℓ does not specifically require integer values of n, it should not matter.

Thus, using our simplified synchronized service approximation, we are able to approximately model utilization, stability and average response times for an arbitrary speculation scenario $\mathcal{D}_d$ replicated over PS queues, as long as the resulting method for extracting $\mathbb{E}[T]$ under ℓ can be obtained with arrival rate λ, service rate $\mu(\mathcal{D}_d)$, and number of servers $n(\mathcal{D}_d)$. However, a drawback of our approach is that it might be complicated to implement triggering of speculative clones at the completed service times limits defined by δ_i, as these can be cumbersome to keep track of in a real system.

As a final note, the speculative execution model can further be used to model standard cloning for which all clones get sent at $\delta_i = 0$. In these cases, (6.25) simply becomes the minimum CDF over all clones as given in Lemma 6.2, and as the probability of a clone being dispatched is 1 and has the sojourn factor of 1, $f^{\lambda}(\mathcal{D}_d)$ will assume an integer value. Furthermore, if modeling the clone-to-cluster strategy, where the n servers are divided into m clusters of d clones each, $f^{\lambda}(\mathcal{D}_d) = d$ and $n(\mathcal{D}_d) = {}^{n}/_{d} = m$. Hence, the clone-to-cluster strategy can be seen as a special case of our speculative execution model, where the model is exact, as clone-to-cluster guarantees synchronized service.

Evaluation via Simulation Experiment

To evaluate our approximation method for speculative execution, we performed a simulation experiment. A system with $n = 10$ servers with $k = 1$ following the PS discipline was assumed. Due to its near-synchronized properties as shown in Figure 6.7, it was further assumed that the servers lie behind a JSQ load balancer. To obtain explicit response time measures, we need to assume that our arrival rates λ to the system are Poisson. This allows us to utilize the aforementioned mean response time approximation for M/G/1-PS queues under JSQ from [Gupta et al., 2007a]. The system was then subjected to one of the following three different speculation scenarios

$$\mathcal{D}_1 = \{1.5\}, \quad \mathcal{D}_2 = \{0.7, 1.0\}, \quad \mathcal{D}_3 = \{0.3, 0.6, 0.9\}. \tag{6.34}$$

It is assumed that the service times on all servers follow a Pareto distribution with $\Phi^s(t) = 1 - \left(\frac{0.5}{t}\right)^{2.1}$ if $t \geq 0.5$ else $\Phi^s(t) = 0$. The shape and scale parameter were chosen such that the mean service time became 1 for all servers. For each of these three speculation scenarios, we considered a utilization interval $\rho(\mathcal{D}_d)$ from 0.3 to 0.9 by properly choosing the Poisson rates λ via (6.32). Then, for each utilization and speculation scenario, we calculated $\rho(\mathcal{D}_d)$ and $\mathbb{E}[T(\mathcal{D}_d)]$ from the approximation method, and compared to $\widehat{\rho}(\mathcal{D}_d)$ and $\widehat{T}(\mathcal{D}_d)$ from 20 simulations with 10^6 requests each.

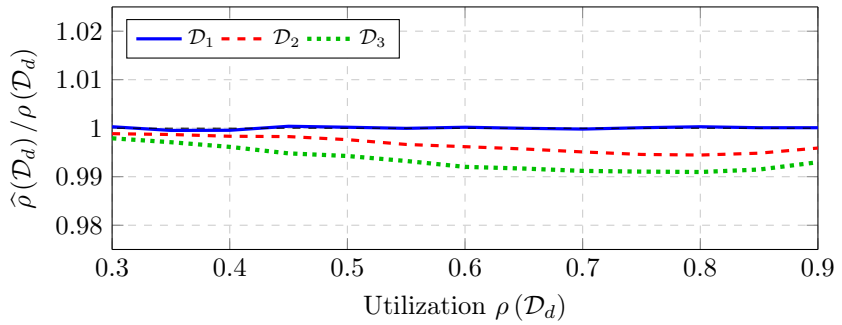
The results can be seen in Figure 6.9. Here, three comparisons between simulated and approximated utilization and mean response times are shown for the scenarios $\mathcal{D}_1$ (blue), $\mathcal{D}_2$ (red dashed) and $\mathcal{D}_3$ (green dotted).

In Figure 6.9(a), the simulated system utilization $\widehat{\rho}(\mathcal{D}_d)$ is normalized against our modeled $\rho(\mathcal{D}_d)$. The results are very close to 1 for all scenarios and loads. Furthermore, in Figure 6.9(b) the results of the simulated average response times $\widehat{T}(\mathcal{D}_d)$ normalized against our modeled $\mathbb{E}[T(\mathcal{D}_d)]$ are shown. As can be seen, the accuracy of our model is high for low to medium loads for all three speculation scenarios. However, as the utilization increases, the accuracy decreases, albeit still within a reasonable level for all three scenarios. The decrease in accuracy is more prominent the more speculative clones the scenario contains, which leads us to believe that a probable explanation is that the system becomes overall less synchronized the more speculative clones it has to deal with. In all, these two figures point towards the fact that our model can be used to accurately predict utilization, stability, and mean response time for systems under speculative execution as long as they are near-synchronized.

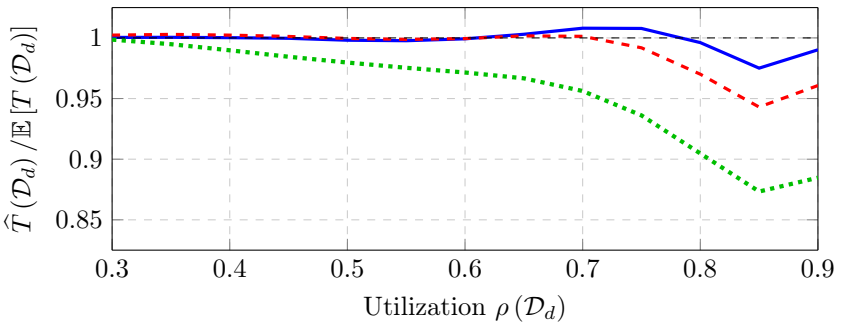
The final Figure 6.9(c) shows the results of the simulated average response times $\widehat{T}(\mathcal{D}_d)$ for the speculation scenarios normalized against $\widehat{T}(\mathcal{D}_0)$, where there is no (speculative) cloning. The aim of this figure is to examine the potential benefits of employing speculative execution under different utilization levels. A value below 1 indicates that a speculation scenario is beneficial and, as can be seen, all three scenarios perform well at low loads. Scenario $\mathcal{D}_1$ distinguishes itself from the others by actually outperforming the no-speculation case at all system loads. This is an interesting contrast to standard cloning (all $s_i = 0$). According to the results presented in the clone-to-all and clone-to-cluster experiments shown in Figure 6.3 and Figure 6.4, pure cloning under this particular Pareto distribution is only beneficial for low loads. Speculative execution thus has the potential to handle request redundancy at higher loads, where standard cloning normally fails. This, however, requires us to correctly identify a viable speculation scenario. In scenario $\mathcal{D}_1$, a speculative clone is sent after 1.5 seconds, which for this particular experiment gives a good trade-off between increasing load and combating the heavy tails of the Pareto distributed service time. Looking at the other two scenarios, a suitable first candidate heuristic could then be to choose a scenario with a single speculative clone where $\delta_1 > \mathbb{E}[t^s]$.

6.5 Summary and Discussion

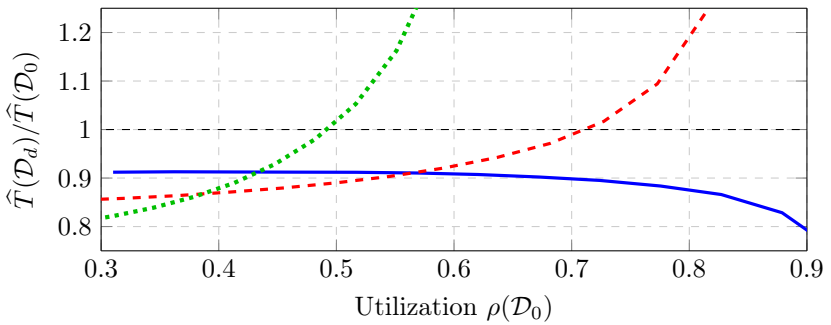
In this chapter, we have presented a theoretical analysis that extends and generalizes known results regarding request cloning in server systems. This was possible due to the introduction of the concept synchronized service,



(a) Simulated vs modeled $\rho(\mathcal{D}_d)$.



(b) Simulated vs modeled $T(\mathcal{D}_d)$.



(c) Speculation scenarios $\mathcal{D}_d$ vs $\mathcal{D}_0$.

Figure 6.9 Comparison of utilizations and mean response time between simulated values and values from the approximation method, for the three speculation scenarios $\mathcal{D}_1$ (blue line), $\mathcal{D}_2$ (red dashed) and $\mathcal{D}_3$ (green dotted). For each scenario and utilization, the simulated values were obtained using 20 repeated runs with 10^6 requests each. The confidence intervals are tight and left out for readability. The legend applies to all figures.

which allows us to model systems that fulfill the required assumptions with an equivalent G/G/k queue. We showed that no assumptions on either interarrival or service time distributions are required, and that the G/G/k model holds for any deterministic queueing discipline. However, the assumptions for synchronized service are unrealistic and limiting. Focusing on the PS discipline, we studied the relaxation of these limitations. In particular, we

- Studied two systems where synchronized service is achieved, clone-to-all and clone-to-cluster, and investigated optimal cloning factors and load balancing codesigns under dependent service time distributions.
- Relaxed the necessary assumptions and studied how clone-to-all is affected by delays, and how clone-to-cluster can be used to approximate a system where the clones are freely assigned (denoted clone-to-any).
- Introduced an extension of the clone-to-cluster strategy to approximately model systems under speculative execution.

We further developed a discrete event simulator and tested the different concepts in a series of simulation experiments.

The simulation campaign shows good compliance with our theoretical findings. For both the clone-to-all plots in Figure 6.3, and the codesign plot in Figure 6.4, the model predicts optimal cloning factors c_f^o and the corresponding mean response time $\mathbb{E}[T]$ with very high accuracy. Furthermore, Figure 6.6 shows that the introduced delays have only a minor influence on $\mathbb{E}[T]$ and, especially for arrival delays, the approximate upper bounds can be used to predict the effect of practical imperfections. Moreover, in Figure 6.7 the interesting near-synchronized service property of the JSQ load balancing policy is shown. It suggests that our model could accurately describe setups involving JSQ, where synchronized service is not guaranteed. Finally, in Figure 6.9 it is shown that our extended model for speculative execution is accurate under the JSQ load balancer. This is expected, as it assumes synchronized service and should thus be accurate for any system that experiences near-synchronization.

Discussion. Although general regarding the involved distributions and queueing discipline, the additional assumptions required make the concept of synchronized service mostly a tool for theoretical analysis. However, concerning systems of PS queues, two key takeaways from this chapter are (i) delays on the arrival and cancellation of clones seem to have only minor effect on the system, and (ii) systems under JSQ load balancing becomes near-synchronized. Hence, since there exists an accurate model for JSQ over M/G/1-PS queues, the equivalent G/G/k model can be used to accurately approximate both cloning and speculative execution in these systems.

We explicitly studied the delays in a clone-to-all system, as it can be directly compared with the equivalent G/G/k model. However, as clone-to-cluster is simply a combination of multiple such systems behind a load balancer, the same low impact of delays should apply. Subsequently, the effects on a near-synchronized clone-to-any system should also be small. Further, the introduction of delays slightly increases the mean response time, but as clone-to-cluster probably forms an upper bound to clone-to-any, using clone-to-cluster to approximate a clone-to-any systems with small delays could even yield an improved accuracy.

7

Thesis Summary

In this thesis, the subject of cloud application performance modeling has been studied. Focus has been put on modeling using queueing theory, especially considering the processor sharing discipline and the fluid model for evaluating the mean queue length dynamics in queueing networks. In particular, the thesis has presented the following contributions.

- In Chapter 3, the thesis improved the mean-field fluid model by extending it to mixed networks of PS and INF queues and introducing a cheap refinement method and a closed-form approximation for the response time CDF.
- In Chapter 4, these improvements were then used to create a simple but general fluid model for microservice applications that can be completely and distributively extracted at runtime from common tracing data.
- In Chapter 5, a method was devised to optimize the cost of a running cloud application under arbitrary costs and performance constraints by tuning the load balancing parameters using automatic differentiation of the introduced microservice fluid model.
- In Chapter 6, it was shown that certain cloning systems can be expressed as a G/G/k queue. In addition, how cloning is affected by relaxing the necessary assumptions and how the method can be used to approximate speculative execution was studied for M/G/1-PS queues.

7.1 Discussion

The most significant contribution of this thesis is, according to its author, the smoothed mean-field fluid model introduced in Section 3.3. The simple inverse p-norm smoothing of the processor share function yields a surprisingly efficient refinement method for mixed networks of PS and INF queues. The

method is shown to greatly reduce the mean-field approximation error for queueing networks of small system size, i.e., small initial population, few servers, and low arrival rates. Furthermore, the method obtains some nice theoretical properties. From data of a stationary system, it is possible to quickly obtain an optimal smoothing value, and the nominal mean-field fluid model is recovered if the smoothing value goes to infinity.

This new fluid model allows important performance metrics to be accurately approximated at low cost for cloud applications modeled as multi-class networks of PS and INF queues with pseudo-general service times. From the mean queue lengths the standard metrics throughput, utilization, and mean response time can readily be obtained. The model also allows for the response time CDF to be quickly approximated over almost any subset of classes. This is important as response time percentiles are prevalent in SLAs but often difficult to model.

To put the smoothed mean-field fluid model to use, we applied it to model cloud applications following the microservice architecture in multi-cloud deployments. In the model, each service replica is represented as a single PS queue and each replica-to-replica delay as a single INF queue. Considering large and complex services, this choice of structure might fail to capture important details, but it makes more sense for microservices, where each service should be simple and handle very few tasks. It is worth noting that single-queue models of services and delays are commonplace. The novelty of the method instead lies in the general forms of delay afflicted service graphs it can capture. Also, the resulting microservice fluid model can be created at runtime from common tracing data in a distributed fashion considering commonly used communication layers, such as service meshes utilizing local proxies. This makes the fluid model simple to obtain and keep up-to-date on the current state of a running application.

Although such simple application models can lack in accuracy, they might be accurate enough when used to design methods for balancing costs and performance. Furthermore, simple models are often quicker to extract and evaluate, and it is paramount that the cost of extraction and running the resulting algorithm is less than the cost it manages to save.

Thanks to modern software tools, it is easy to design such methods from our microservice fluid model via automatic differentiation. This makes it possible to obtain the gradient of an arbitrary cost function over performance metrics obtainable from the fluid model. In particular, this gradient can then be used to create a cost optimizing algorithm that adhere to, e.g., response time percentile constraints. In the thesis, this is exemplified by tuning load balancing parameters, but the method can potentially be applied to all manners of actions as long as they can be expressed in the fluid model.

Using our FedApp sandbox, the microservice fluid model and the cost optimizing algorithm were evaluated on a small microservice application de-

ployed over multiple Kubernetes clusters with emulated cluster-to-cluster network characteristics. Although the sandbox greatly simplifies the effort required to set up the necessary environment to perform such experiments, its use cases naturally go beyond evaluating performance models in multi-cloud environments.

Regarding the results on redundant requests; although an interesting concept in itself, the use cases of synchronized service and the equivalent G/G/k model will be limited due to the unrealistic assumptions required. However, given a cloning system of PS queues, we showed that relaxing these assumptions by (i) introducing delays in the arrival and cancellation of clones and (ii) allowing clones to be freely placed on a subset of the servers via the JSQ load balancer only had a minor effect. The equivalent G/G/k queue can thus be used to approximately model cloning for these cases.

Threats to validity and potential criticism. The thesis lacks in direct comparison to other similar methods and evaluation over more advanced systems. For example, the smoothed mean-field fluid model could have been compared to approximate MVA methods for multi-server PS queues such as [Kattepur and Nambiar, 2015; Casale et al., 2015]. However, this was not prioritized as the fluid model obtained a good performance in the simulations. How this holds up for more complicated networks is unclear, and thus larger experimental evaluations are warranted.

With respect to the microservice fluid model and the cost optimizing algorithm, a major missing piece is the lack of testing on more advanced microservice applications. Furthermore, the fluid model should preferably have been compared to other existing microservice application models, such as the LQN model of the sockshop microsevice demo[1] used in [Gias et al., 2019]. We started down this path in an early iteration of our work, but found that some services in the examined benchmarks were difficult to get to function properly, lacked robustness at higher utilization, and were hard to adapt to our FedApp testbed. Thus, it was deemed more time efficient to create our own simple experimental example application from scratch.

Further considering the cost optimizing algorithm, the gradient can be obtained quite fast for the simplified example application. However, the current implementation scales poorly with increasing parameters and model size, as seen for the full example application. In order for the approach to be applicable in more advanced settings, it is important that this can be remedied by, e.g., tuning the evaluation speed of the cost function or using other methods for automatic differentiation than forward-mode.

Furthermore, all three chapters using the fluid model lack proper evaluation on transient performance metrics, even though this is one of the key features of the fluid model. In the thesis, this is motivated by the fact that

[1] https://github.com/microservices-demo/microservices-demo

it is cumbersome to make a good comparison to transient metrics from experiments because the systems we consider are highly stochastic. Also, the experiments performed in Chapter 3 seem to suggest that the transients of the mean-field fluid model are well-behaved. This should however be investigated more carefully.

Considering the results on request cloning, the thesis lacks evaluation on a real system, especially with regard to the validity of synchronizing the cloned requests and how it could be implemented. Furthermore, the notion of cost is not taken into account. For example, to lower costs in public clouds, it is often desired to maximize utilization by scaling down. In these high utilization settings, cloning every request would, as shown, probably not be beneficial regarding the mean response time. However, speculative execution can still be beneficial. Moreover, pure cloning could potentially be used to reduce response times if constraints are violated at a low utilization. Finally, cloning every request at high utilization could still be beneficial with regards to the response time percentiles, but this is not considered in the thesis.

7.2 Future Work

In this section, some possible directions for future work are given. The most important path forward is to address the previous criticism, especially the lack of experiments on more advanced microservice applications.

Apart from that, one important direction is to investigate the potential to combine the results on the mean-field fluid model with the results on request cloning. Fork-join systems are in general difficult to model via the mean-field approximation, as joining destroys the necessary Markov property, but cloning and speculative execution could potentially be included via single-queue approximations. For example, if a PS queueing network contains one or more cloning systems, these systems could potentially be replaced with their corresponding equivalent G/G/k model. The service times in these G/G/k models could then be approximated with a PH distribution, and in turn the mean-field fluid model applied.

Each chapter further has several directions that are worth pursuing, which are summarized below.

The smoothed mean-field fluid model. Regarding the results on the mean-field fluid model from Chapter 3, it would be interesting to see these results extended to other queueing disciplines previously used in mean-field approximations. Such disciplines includes, e.g., single-class M/M/k-FCFS or discriminatory PS with different class priorities as seen in [Zhu et al., 2020].

In addition, a larger study on how to adapt fork-join systems for the mean-field fluid model would be of importance. As discussed, cloning could potentially be included by utilizing a single-queue approximation. Perhaps

something similar could be used for general (n, k) fork-join systems, by replacing the k'th minimum over n service times with a PH distribution in some clever way.

Furthermore, additional work is needed on how the smoothing parameters fitted at the current operating condition affect distant predictions and how they can be corrected to increase accuracy. For example, Section 4.4 seems to suggest that the current method of fitting the smoothing parameters will yield a poor prediction for perturbations that increases the utilization.

Finally, it is valuable that the mean-field fluid model can be used to approximate response time percentiles, but disheartening that the accuracy gets worse the larger the queue length variability becomes. This makes it difficult to accurately predict percentiles in certain important scenarios, e.g., at high utilization under Poisson arrivals. Thus, ways of improving the response time percentile approximation are of importance to study.

The microservice fluid model. In addition to the directions for the smoothed mean-field fluid model, there are some particular paths for future work for the microservice fluid model introduced in Chapter 4.

First and foremost, to increase the generality of the model, it is of great importance that it can handle situations where the services process requests in a manner that is highly different from the PS discipline. This is closely related to testing the fluid model on more advanced microservice applications. In doing so, it is possible to discover when the model holds and when it breaks and how it might be extended to accommodate these situations.

Furthermore, it is important to handle situations where the service time distributions are affected by scaling. This would enable predictions over vertical scaling, and horizontal scaling or migration to new locations with different service rates. One potential strategy to investigate is how scaling of the existing PH distributions could be applied in these situations.

Scaling of the PH distributions could potentially also be used to generate a more efficient model tracking, instead of extracting a completely new model at each sampling time. As obtaining the service times is quite efficient, one could, for example, extract the distributions once and then track their first moments using a moving average filter. These moments could then be used to scale the initial distributions. It might even be possible to track the PH distributions without having to measure every arrival and departure by utilizing a Kalman filter together with measurements on response time mean and percentiles to update the distribution parameters.

Finally, it is limiting that our fluid model needs to assume a random load balancer, and thus it is important to investigate how other strategies could be incorporated. For example, considering round robin, it might be enough to directly approximate the strategy with the random strategy. Furthermore, considering JSQ the accurate approximation for PS queues in [Gupta et al.,

2007b] could potentially be used, as it reduces such load balancing systems to a single M/M/1-PS queue with a state dependent arrival rate.

The cost optimizing algorithm. The choice to focus on optimizing the load balancing weights using simple gradient stepping in Chapter 5 was made to exemplify the procedure. We wanted a problem whose fluid model representation was simple, had a comprehensible control variable update, and was relatively easy to validate experimentally. However, the generality of automatic differentiation allows us to consider a whole range of other ways to optimize a running application, e.g., scaling and migration. As long as such an action can be represented in the microservice fluid model. Also, more advanced optimization methods could be used to update the control variables. Multiple prediction steps in some form of nonlinear MPC approach or basing the next step on the optimum of the current fitted model are two examples that could be considered. Furthermore, automatic differentiation allows us to consider methods that use higher-order derivatives.

It would be interesting to consider systems where the transients play a larger part. In our example application, these quickly diminish, and the system enters stationarity in a short time frame. To consider transients, the adaptation speed of the model extraction needs to be increased. The current method of gathering data from the application over some sample time is simply not fast enough. In fact, the dynamics seen in, e.g., Figure 5.6 is entirely based on our mistrust of the model and incremental parameter updates. Instead, the extraction could be replaced by some quicker model tracking as discussed, and the parameters could be updated in smaller steps in a time frame that is more in line with the transients.

Redundant requests. The assumptions introduced to obtain synchronized service in Chapter 6 are limiting, but the resulting equivalent G/G/k model can be used as an adequate approximation when relaxing the assumptions for certain cloning systems of PS queues. How these relaxations hold up to other queueing disciplines would be interesting to consider.

We theorize that for the INF discipline, as all requests are served according to Φ^s directly, a system should always be synchronized under both cloning and speculative execution. Cancellation delays should not have any effect, but arrival delays should still affect the mean response time. The FCFS discipline, on the other hand, as it has no processing of requests until the queue in front is cleared, should experience a more severe effect from both delays and clone-to-any.

The concept of synchronization error could further be expanded, as it is only briefly discussed with respect to relaxing the clone-to-all assumption. Instead, the concept could be studied in more general settings to create intuition on which situations synchronized service could yield an adequate approximation. For example, the synchronization error obtained for arrival

and cancellation delays could be examined.

The chapter only considers mean response times and no percentiles, although cloning or speculative execution is often motivated by reducing tail latency. In certain cases, e.g., considering PS queues under Poisson arrivals and $k = 1$, closed-form expressions of the response time distribution exist. Otherwise, if one is able to combine the fluid model with request cloning, as discussed, then this can yield another way of analyzing the percentiles.

Bibliography

Abreu, D. P., K. Velasquez, M. Curado, and E. Monteiro (2019). "A comparative analysis of simulators for the cloud to fog continuum". *Simulation Modelling Practice and Theory*, p. 102029. DOI: 10.1016/j.simpat.2019.102029.

Addis, B., D. Ardagna, B. Panicucci, M. S. Squillante, and L. Zhang (2013). "A hierarchical approach for the resource management of very large cloud platforms". *IEEE Transactions on Dependable and Secure Computing* **10**:5, pp. 253–272. DOI: 10.1109/TDSC.2013.4.

Ahmed, A. and A. S. Sabyasachi (2014). *Cloud computing simulators: a detailed survey and future direction*. DOI: 10.1109/IAdCC.2014.6779436.

Aktaş, M. F. and E. Soljanin (2019). "Straggler mitigation at scale". *IEEE/ACM Transactions on Networking* **27**:6, pp. 2266–2279. DOI: 10.1109/TNET.2019.2946464.

Amiri, M. and L. Mohammad-Khanli (2017). "Survey on prediction models of applications for resources provisioning in cloud". *Journal of Network and Computer Applications* **82**, pp. 93–113. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.01.016.

Ananthanarayanan, G., A. Ghodsi, S. Shenker, and I. Stoica (2013). "Effective straggler mitigation: attack of the clones". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*. USENIX Association, Lombard, IL, pp. 185–198. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ananthanarayanan.

Anselmi, J. and G. Casale (2013). "Heavy-traffic revenue maximization in parallel multiclass queues". *Performance Evaluation* **70**:10. Proceedings of IFIP Performance 2013 Conference, pp. 806–821. ISSN: 0166-5316. DOI: 10.1016/j.peva.2013.08.008.

Anselmi, J., B. D'Auria, and N. Walton (2013). "Closed queueing networks under congestion: nonbottleneck independence and bottleneck convergence". *Mathematics of Operations Research* **38**:3, pp. 469–491. DOI: 10.1287/moor.1120.0583.

Ardagna, D., G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang (2014). "Quality-of-service in cloud computing: modeling techniques and their applications". en. *Journal of Internet Services and Applications* **5**:1, p. 11. ISSN: 1869-0238. DOI: 10.1186/s13174-014-0011-3.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, et al. (2009). *Above the clouds: A berkeley view of cloud computing*. Tech. rep. Technical Report UCB/EECS-2009-28, EECS Department, University of California at Berkeley.

Asmussen, S., O. Nerman, and M. Olsson (1996). "Fitting phase-type distributions via the em algorithm". *Scandinavian Journal of Statistics* **23**:4, pp. 419–441. ISSN: 03036898, 14679469. URL: http://www.jstor.org/stable/4616418.

Awad, M. and D. A. Menascé (2020). "Imodel: automatic derivation of analytic performance models". **5**:2. ISSN: 2376-3639. DOI: 10.1145/3374220.

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.

Azar, Y., A. Z. Broder, A. R. Karlin, and E. Upfal (1994). "Balanced allocations". In: *Proceedings of the twenty-sixth annual ACM symposium on theory of computing*, pp. 593–602.

Balter, M. (2013). *Performance modeling and design of computer systems : queueing theory in action*. Cambridge University Press, Cambridge. ISBN: 9781107027503.

Barna, C., H. Khazaei, M. Fokaefs, and M. Litoiu (2017). "Delivering elastic containerized cloud applications to enable devops". In: *IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '17)*, pp. 65–75. DOI: 10.1109/SEAMS.2017.12.

Barna, C., M. Litoiu, M. Fokaefs, M. Shtern, and J. Wigglesworth (2018). "Runtime performance management for cloud applications with adaptive controllers". In: *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pp. 176–183.

Baskett, F., K. M. Chandy, R. R. Muntz, and F. G. Palacios (1975). "Open, closed, and mixed networks of queues with different classes of customers". *J. ACM* **22**:2, pp. 248–260. ISSN: 0004-5411. DOI: 10.1145/321879.321887.

Becker, S., H. Koziolek, and R. Reussner (2009). "The palladio component model for model-driven performance prediction". *J. Systems and Software* **82**:1, pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

Begin, T. and A. Brandwajn (2016). "Predicting the system performance by combining calibrated performance models of its components: a preliminary study". In: *7th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '16)*. Association for Computing Machinery, Delft, The Netherlands, pp. 95–100. ISBN: 9781450340809. DOI: 10.1145/2851553.2858658.

Benaïm, M. and J.-Y. Le Boudec (2008). "A class of mean field interaction models for computer and communication systems". *Performance Evaluation* **65**:11. Performance Evaluation Methodologies and Tools: Selected Papers from ValueTools 2007, pp. 823–838. ISSN: 0166-5316. DOI: 10.1016/j.peva.2008.03.005.

Bertoli, M., G. Casale, and G. Serazzi (2009). "Jmt: performance engineering tools for system modeling". *SIGMETRICS Perform. Eval. Rev.* **36**:4, pp. 10–15. ISSN: 0163-5999. DOI: http://doi.acm.org/10.1145/1530873.1530877.

Beyer, B., C. Jones, J. Petoff, and N. R. Murphy (2016). *Site reliability engineering: How Google runs production systems*. O'Reilly Media, Inc.

Bolch, G., S. Greiner, H. de Meer, and K. S. Trivedi (2006). *Queueing Networks and Markov Chains*. John Wiley & Sons, Inc., Hoboken, N.J. ISBN: 9780471791577. DOI: 10.1002/0471791571.

Borst, S. C. (1995). "Optimal probabilistic allocation of customer types to servers". **23**:1, pp. 116–125. ISSN: 0163-5999. DOI: 10.1145/223586.223601.

Bortolussi, L., J. Hillston, D. Latella, and M. Massink (2013). "Continuous approximation of collective system behaviour: a tutorial". *Performance Evaluation* **70**:5, pp. 317–349. ISSN: 0166-5316. DOI: 10.1016/j.peva.2013.01.001.

Bramson, M. (1996). "Convergence to equilibria for fluid models of head-of-the-line proportional processor sharing queueing networks". *Queueing Systems* **23**:1-4, pp. 1–26. DOI: 10.1007/bf01206549.

Bramson, M., Y. Lu, and B. Prabhakar (2010). "Randomized load balancing with general service time distributions". *SIGMETRICS Perform. Eval. Rev.* **38**:1, pp. 275–286. ISSN: 0163-5999. DOI: 10.1145/1811099.1811071.

Braverman, A. and J. G. Dai (2017). "Stein's method for steady-state diffusion approximations of M/PH/n+M systems". *The Annals of Applied Probability* **27**:1, pp. 550–581. DOI: 10.1214/16-AAP1211.

Brogi, A., S. Corfini, and S. Iardella (2007). "From owl-s descriptions to petri nets". In: *International Conference on Service-Oriented Computing*. Springer, pp. 427–438. DOI: 10.1007/978-3-540-93851-4_41.

Brosig, F., N. Huber, and S. Kounev (2011). "Automated extraction of architecture-level performance models of distributed component-based systems". In: *26th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '11)*, pp. 183–192. DOI: 10.1109/ASE.2011.6100052.

Buyya, R., S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. S. Netto, A. N. Toosi, M. A. Rodriguez, I. M. Llorente, S. D. C. D. Vimercati, P. Samarati, D. Milojicic, C. Varela, R. Bahsoon, M. D. D. Assuncao, O. Rana, W. Zhou, H. Jin, W. Gentzsch, A. Y. Zomaya, and H. Shen (2018). "A manifesto for future generation cloud computing: research directions for the next decade". *ACM Comput. Surv.* **51**:5. ISSN: 0360-0300. DOI: 10.1145/3241737.

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". *Software: Practice and Experience* **41**:1, pp. 23–50. DOI: 10.1002/spe.995.

Cao, J., M. Andersson, C. Nyberg, and M. Kihl (2003). "Web server performance modeling using an m/g/1/k*ps queue". In: *10th International Conference on Telecommunications, 2003. ICT 2003.* Vol. 2, 1501–1506 vol.2. DOI: 10.1109/ICTEL.2003.1191656.

Casale, G. (2010). "Approximating passage time distributions in queueing models by bayesian expansion". *Performance Evaluation* **67**:11. Performance 2010, pp. 1076–1091. ISSN: 0166-5316. DOI: 10.1016/j.peva.2010.08.003.

Casale, G. (2020). "Integrated performance evaluation of extended queueing network models with line". In: *2020 Winter Simulation Conference (WSC '20)*, pp. 2377–2388. DOI: 10.1109/WSC48552.2020.9383931.

Casale, G., J. F. Pérez, and W. Wang (2015). "Qd-amva: evaluating systems with queue-dependent service requirements". *Performance Evaluation* **91**. Special Issue: Performance 2015, pp. 80–98. ISSN: 0166-5316. DOI: 10.1016/j.peva.2015.06.006.

Cerny, T., M. J. Donahoo, and M. Trnka (2018). "Contextual understanding of microservice architecture: current and future directions". **17**:4, pp. 29–45. ISSN: 1559-6915. DOI: 10.1145/3183628.3183631.

Cox, D. R. (1955). "A use of complex probabilities in the theory of stochastic processes". *Mathematical Proceedings of the Cambridge Philosophical Society* **51**:2, pp. 313–319. DOI: 10.1017/S0305004100030231.

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.

Desnoyers, P., T. Wood, P. Shenoy, R. Singh, S. Patil, and H. Vin (2012). "Modellus: automated modeling of complex internet data center applications". *ACM Trans. Web* **6**:2. ISSN: 1559-1131. DOI: 10.1145/2180861.2180865.

Duan, Q. (2011). "Modeling and performance analysis on network virtualization for composite network-cloud service provisioning". In: *2011 IEEE World Congress on Services*, pp. 548–555. DOI: 10.1109/SERVICES.2011.10.

Eastham, M. S. P. (1961). "2968. On the Definition of Dual Numbers". *The Mathematical Gazette* **45**:353, pp. 232–233. ISSN: 0025-5572. DOI: 10.2307/3612794.

Faisal, A., D. Petriu, and M. Woodside (2013). "Network latency impact on performance of software deployed across multiple clouds". In: *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '13)*. IBM Corp., Ontario, Canada, pp. 216–229.

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". *ACM Transactions on Autonomous and Adaptive Systems* **11**:4, pp. 1–31. DOI: 10.1145/3024188.

Franks, G., T. Al-Omari, M. Woodside, O. Das, and S. Derisavi (2009). "Enhanced modeling and solution of layered queueing networks". *IEEE Transactions on Software Engineering* **35**:2, pp. 148–161. DOI: 10.1109/TSE.2008.74.

Fratta, L., M. Gerla, and L. Kleinrock (1973). "The flow deviation method: an approach to store-and-forward communication network design". *Networks* **3**:2, pp. 97–133. DOI: 10.1002/net.3230030202.

Gan, Y., Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, and 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 (ASPLOS '19)*. Association for Computing Machinery, Providence, RI, USA, pp. 3–18. ISBN: 9781450362405. DOI: 10.1145/3297858.3304013.

Garbi, G., E. Incerto, and M. Tribastone (2020). "Learning queuing networks by recurrent neural networks". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, Edmonton AB, Canada, pp. 56–66. ISBN: 9781450369916. DOI: 10.1145/3358960.3379134.

Gardiner, C. (1985). *Handbook of stochastic methods - for physics, chemistry and the natural sciences, Second Edition*. Springer-Verlag, Berlin New York.

Gardner, K. (2017). *Modeling and Analyzing Systems with Redundancy*. PhD thesis. Carnegie Mellon University, Pittsburgh, PA 15213.

Gardner, K., M. Harchol-Balter, and A. Scheller-Wolf (2016a). "A better model for job redundancy: decoupling server slowdown and job size". In: *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '16)*. DOI: 10.1109/mascots.2016.43.

Gardner, K., M. Harchol-Balter, A. Scheller-Wolf, M. Velednitsky, and S. Zbarsky (2017). "Redundancy-d: the power of d choices for redundancy". *Operations Research* **65**:4, pp. 1078–1094. DOI: 10.1287/opre.2016.1582.

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. DOI: 10.1145/2796314.2745873.

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.

Gast, N. (2020). *Refinements of Mean Field Approximation*. Tech. rep. Grenoble. URL: https://tel.archives-ouvertes.fr/tel-02509756.

Gast, N. and B. Van Houdt (2017). "A refined mean field approximation". *Proc. ACM Meas. Anal. Comput. Syst.* **1**:2. DOI: 10.1145/3154491.

Gias, A. U., G. Casale, and M. Woodside (2019). "Atom: model-driven autoscaling for microservices". In: *IEEE 39th International Conference on Distributed Computing Systems (ICDCS '19)*, pp. 1994–2004. DOI: 10.1109/ICDCS.2019.00197.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Goudarzi, H. and M. Pedram (2011). "Multi-dimensional sla-based resource allocation for multi-tier cloud computing systems". In: *IEEE 4th International Conference on Cloud Computing (CLOUD '11)*, pp. 324–331. DOI: 10.1109/CLOUD.2011.106.

Guo, X., Y. Lu, and M. S. Squillante (2004). "Optimal probabilistic routing in distributed parallel queues". *SIGMETRICS Perform. Eval. Rev.* **32**:2, pp. 53–54. ISSN: 0163-5999. DOI: 10.1145/1035334.1035355.

Gupta, V., M. H. Balter, K. Sigman, and W. Whitt (2007a). "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.

Gupta, V., M. Harchol Balter, K. Sigman, and W. Whitt (2007b). "Analysis of join-the-shortest-queue routing for web server farms". *Performance Evaluation* **64**:9. Performance 2007, pp. 1062–1081. ISSN: 0166-5316. DOI: 10.1016/j.peva.2007.06.012.

Harbaoui, A., N. Salmi, B. Dillenseger, and J. Vincent (2010). "Introducing queuing network-based performance awareness in autonomic systems". In: *6th Int'l Conf. on Autonomic and Autonomous Systems*, pp. 7–12. DOI: 10.1109/ICAS.2010.9.

Harrison, P. G. and W. J. Knottenbelt (2006). "Quantiles of sojourn times". *Computer System Performance Modeling In Perspective: A Tribute to the Work of Prof Kenneth C Sevcik*, pp. 155–193. DOI: 10.1142/9781860948923_0010.

Heimerson, A., R. Brännvall, J. Sjölund, J. Eker, and J. Gustafsson (2021). "Towards a holistic controller: reinforcement learning for data center control". In: *Proceedings of the Twelfth ACM International Conference on Future Energy Systems (e-Energy '21)*. Association for Computing Machinery, Virtual Event, Italy, pp. 424–429. ISBN: 9781450383332. DOI: 10.1145/3447555.3466581.

Heimerson, A., J. Ruuskanen, and J. Eker (2022). "Automatic differentiation over fluid models for holistic load balancing". In: *IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS '22), to appear.*

Hermanns, H., U. Herzog, and J.-P. Katoen (2002). "Process algebra for performance evaluation". *Theoretical Computer Science* **274**:1. Ninth International Conference on Concurrency Theory 1998, pp. 43–87. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(00)00305-4.

Hindmarsh, A. C. and L. R. Petzold (2005). *LSODA, Ordinary Differential Equation Solver for Stiff or Non-Stiff System*. Tech. rep. Nuclear Energy Agency of the OECD (NEA). URL: http://www.nea.fr/abs/html/uscd1227.html.

Hordijk, A. and J. Loeve (2000). "Optimal static customer routing in a closed queuing network". *Statistica Neerlandica* **54**:2, pp. 148–159. DOI: 10.1111/1467-9574.00133.

Hrischuk, C., C. Murray Woodside, and J. Rolia (1999). "Trace-based load characterization for generating performance software models". *IEEE Transactions on Software Engineering* **25**:1, pp. 122–135. DOI: 10.1109/32.748921.

Incerto, E., A. Napolitano, and M. Tribastone (2018a). "Moving horizon estimation of service demands in queuing networks". In: *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '18)*. IEEE Computer Society, Los Alamitos, CA, USA, pp. 348–354. DOI: 10.1109/MASCOTS.2018.00040.

Incerto, E., A. Napolitano, and M. Tribastone (2021). "Learning queuing networks via linear optimization". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '21)*. Association for Computing Machinery, Virtual Event, France, pp. 51–60. ISBN: 9781450381949. DOI: 10.1145/3427921.3450245.

Incerto, E. and M. Tribastone (2019). "Model-based Performance Self-adaptation: A Tutorial". In: *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*. Association for Computing Machinery, New York, NY, USA, pp. 49–52. ISBN: 978-1-4503-6286-3. DOI: 10.1145/3302541.3310293.

Incerto, E., M. Tribastone, and C. Trubiani (2017). "Software performance self-adaptation through efficient model predictive control". In: *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE '17)*, pp. 485–496. DOI: 10.1109/ASE.2017.8115660.

Incerto, E., M. Tribastone, and C. Trubiani (2018b). "Combined vertical and horizontal autoscaling through model predictive control". In: Aldinucci, M. et al. (Eds.). *Euro-Par 2018: Parallel Processing*. Springer International Publishing, Cham, pp. 147–159. ISBN: 978-3-319-96983-1. DOI: 10.1007/978-3-319-96983-1_11.

Israr, T. A., D. H. Lau, G. Franks, and M. Woodside (2005). "Automatic generation of layered queuing software performance models from commonly available traces". In: *Proceedings of the 5th International Workshop on Software and Performance (WOSP '05)*. Association for Computing Machinery, Palma, Illes Balears, Spain, pp. 147–158. ISBN: 1595930876. DOI: 10.1145/1071021.1071037.

Jackson, J. R. (1957). "Networks of waiting lines". *Operations research* **5**:4, pp. 518–521. DOI: 10.1287/opre.5.4.518.

Jain, V. and E. Learned-Miller (2010). *FDDB: A Benchmark for Face Detection in Unconstrained Settings*. Tech. rep. UM-CS-2010-009. University of Massachusetts, Amherst. URL: http://vis-www.cs.umass.edu/fddb/.

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 (ICAC '17)*, pp. 135–144. DOI: 10.1109/ICAC.2017.17.

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.

Joshi, G., Y. Liu, and E. Soljanin (2012). "Coding for fast content download". In: *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton '12)*, pp. 326–333. DOI: 10.1109/Allerton.2012.6483236.

Joshi, G., E. Soljanin, and G. Wornell (2015). "Efficient replication of queued tasks for latency reduction in cloud systems". In: *53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton '15)*. IEEE. DOI: 10.1109/allerton.2015.7446992.

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.

Kattepur, A. and M. Nambiar (2015). "Performance modeling of multi-tiered web applications with varying service demands". In: *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW '15)*, pp. 415–424. DOI: 10.1109/IPDPSW.2015.28.

Kielanski, G. and B. Van Houdt (2021). "On the asymptotic insensitivity of the supermarket model in processor sharing systems". *Proc. ACM Meas. Anal. Comput. Syst.* **5**:2. DOI: 10.1145/3460089.

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 '14)*. Association for Computing Machinery, Hyderabad, India, pp. 700–711. ISBN: 9781450327565. DOI: 10.1145/2568225.2568227.

Kobayashi, H. and M. Gerla (1983). "Optimal routing in closed queuing networks". *ACM Transactions on Computer Systems (TOCS)* **1**:4, pp. 294–310.

Koziolek, H. and R. Reussner (2008). "A model transformation from the palladio component model to layered queueing networks". In: *SPEC International Performance Evaluation Workshop*. Springer, pp. 58–78. DOI: 10.1007/978-3-540-69814-2_6.

Kurtz, T. G. (1970). "Solutions of ordinary differential equations as limits of pure jump markov processes". *Journal of Applied Probability* **7**:1, pp. 49–58. DOI: 10.2307/3212147.

Latouche, G. and V. Ramaswami (1999). *Introduction to Matrix Analytic Methods in Stochastic Modeling*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719734.

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.

Lee, R. and B. Jeng (2011). "Load-balancing tactics in cloud". In: *2011 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pp. 447–454. DOI: 10.1109/CyberC.2011.79.

Little, J. D. (1961). "A proof for the queuing formula: l= λ w". *Operations research* **9**:3, pp. 383–387. DOI: 10.1287/opre.9.3.383.

Lu, Y., Q. Xie, G. Kliot, A. Geller, J. R. Larus, and A. Greenberg (2011). "Join-idle-queue: a novel load balancing algorithm for dynamically scalable web services". *Performance Evaluation* **68**:11. Special Issue: Performance 2011, pp. 1056–1071. ISSN: 0166-5316. DOI: 10.1016/j.peva.2011.07.015.

Mandelbaum, A. and G. Pats (1998). "State-dependent stochastic networks. Part I. Approximations and applications with continuous diffusion limits". *The Annals of Applied Probability* **8**:2, pp. 569–646. DOI: 10.1214/aoap/1028903539.

Mell, P., T. Grance, et al. (2011). *The NIST definition of cloud computing*. Tech. rep. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

Merris, R. (1994). "Laplacian matrices of graphs: a survey". *Linear Algebra and its Applications* **197-198**, pp. 143–176. ISSN: 0024-3795. DOI: 10.1016/0024-3795(94)90486-3.

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 '20)*, pp. 196–205. DOI: 10.1109/UCC48980.2020.00038.

Mizan, A. and G. Franks (2011). "An automatic trace based performance evaluation model building for parallel distributed systems". In: *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering (ICPE '11)*. Association for Computing Machinery, Karlsruhe, Germany, pp. 61–72. ISBN: 9781450305198. DOI: 10.1145/1958746.1958760.

Modica, G. and L. Poggiolini (2012). *A First Course in Probability and Markov Chains*. John Wiley & Sons, Ltd. DOI: 10.1002/9781118477793.

Moghaddam, S. K., R. Buyya, and K. Ramamohanarao (2019). "Performance-aware management of cloud resources: a taxonomy and future directions". *ACM Comput. Surv.* **52**:4. ISSN: 0360-0300. DOI: 10.1145/3337956.

Niu, Y., F. Liu, and Z. Li (2018). "Load balancing across microservices". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 198–206. DOI: 10.1109/INFOCOM.2018.8486300.

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020a). "Modeling of request cloning in cloud server systems using processor sharing". In: *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, Edmonton AB, Canada, pp. 24–35. ISBN: 9781450369916. DOI: 10.1145/3358960.3379128.

Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020b). "Towards performance modeling of speculative execution for cloud applications". In: *Companion of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, Edmonton AB, Canada, pp. 17–19. ISBN: 9781450371094. DOI: 10.1145/3375555.3384379.

Nylander, T., M. Thelander Andrén, K.-E. Årzén, and M. Maggio (2018). "Cloud application predictability through integrated load-balancing and service time control". In: *2018 IEEE International Conference on Autonomic Computing (ICAC '18)*, pp. 51–60. DOI: 10.1109/ICAC.2018.00015.

Osogami, T. and M. Harchol-Balter (2006). "Closed form solutions for mapping general distributions to quasi-minimal ph distributions". *Performance Evaluation* **63**:6. Modelling Techniques and Tools for Computer Performance Evaluation, pp. 524–552. ISSN: 0166-5316. DOI: 10.1016/j.peva.2005.06.002.

Pacifici, G., M. Spreitzer, A. N. Tantawi, and A. Youssef (2005). "Performance management for cluster-based web services". *IEEE journal on selected areas in communications* **23**:12, pp. 2333–2343. DOI: 10.1007/978-0-387-35674-7_29.

Perez, J. F., G. Casale, and S. Pacheco-Sanchez (2015). "Estimating computational requirements in multi-threaded applications". *IEEE Trans. Software Engineering* **41**:3, pp. 264–278. DOI: 10.1109/tse.2014.2363472.

Pérez, J. F. and G. Casale (2013). "Assessing sla compliance from palladio component models". In: *IEEE 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '13)*, pp. 409–416. DOI: 10.1109/SYNASC.2013.60.

Pérez, 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.

Qiu, Z., J. F. Perez, R. Birke, L. Chen, and P. G. Harrison (2017). "Cutting latency tail: analyzing and validating replication without canceling". *IEEE Transactions on Parallel and Distributed Systems* **28**:11, pp. 3128–3141. DOI: 10.1109/tpds.2017.2706268.

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. DOI: 10.1145/2851553.2851562.

Qu, C., R. N. Calheiros, and R. Buyya (2018). "Auto-scaling web applications in clouds: a taxonomy and survey". *ACM Comput. Surv.* **51**:4. ISSN: 0360-0300. DOI: 10.1145/3148149.

Raaijmakers, Y., S. Borst, and O. Boxma (2019). "Redundancy scheduling with scaled bernoulli service requirements". *Queueing Systems* **93**:1-2, pp. 67–82. DOI: 10.1007/s11134-019-09621-2.

Rackauckas, C., Y. Ma, J. Martensen, C. Warner, K. Zubov, R. Supekar, D. Skinner, and A. J. Ramadhan (2020). "Universal differential equations for scientific machine learning". *CoRR* **abs/2001.04385**. arXiv: 2001.04385.

Rackauckas, C. and Q. Nie (2017). "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia". *Journal of Open Research Software* **5**:1, p.15.

Rall, L. B. (1981). *Automatic differentiation: Techniques and applications*. Springer.

Randone, F., L. Bortolussi, and M. Tribastone (2021). "Refining mean-field approximations by dynamic state truncation". **5**:2. DOI: 10.1145/3460092.

Reiser, M. and S. S. Lavenberg (1980). "Mean-value analysis of closed multichain queuing networks". *J. ACM* **27**:2, pp. 313–322. ISSN: 0004-5411. DOI: 10.1145/322186.322195.

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)*. Association for Computing Machinery, London, United Kingdom, pp. 379–392. ISBN: 9781450335423. DOI: 10.1145/2785956.2787481.

Revels, J., M. Lubin, and T. Papamarkou (2016). "Forward-Mode Automatic Differentiation in Julia". *arXiv:1607.07892 [cs]*. arXiv: 1607.07892 [cs].

Robertsson, A., B. Wittenmark, M. Kihl, and M. Andersson (2004). "Design and evaluation of load control in web server systems". In: *Proceedings of the 2004 American Control Conference*. Vol. 3, 1980–1985 vol.3. DOI: 10.23919/ACC.2004.1383750.

Rossi, F., M. Nardelli, and V. Cardellini (2019). "Horizontal and vertical scaling of container-based applications using reinforcement learning". In: *IEEE 12th International Conference on Cloud Computing (CLOUD '19)*, pp. 329–338. DOI: 10.1109/CLOUD.2019.00061.

Ruuskanen, J., T. Berner, K.-E. Årzén, and A. Cervin (2021a). "Improving the mean-field fluid model of processor sharing queueing networks for dynamic performance models in cloud computing". *Performance Evaluation* **151**, p. 102231. ISSN: 0166-5316. DOI: https://doi.org/10.1016/j.peva.2021.102231.

Ruuskanen, J. and A. Cervin (2022). "Distributed online extraction of a fluid model for microservice applications using local tracing data". In: *IEEE 15th International Conference on Cloud Computing (CLOUD '22)*, pp. 179–190. DOI: 10.1109/CLOUD55607.2022.00037.

Ruuskanen, J., H. Peng, A. Åkesson, L. Larsson, and M. Kihl (2021b). "FedApp: a research sandbox for application orchestration in federated clouds using OpenStack". *arXiv preprint*. DOI: 10.48550/ARXIV.2109.01480.

Schwarz, J. A., G. Selinka, and R. Stolletz (2016). "Performance analysis of time-dependent queueing systems: survey and classification". *Omega* **63**, pp. 170–189. ISSN: 0305-0483. DOI: 10.1016/j.omega.2015.10.013.

Shah, N. B., K. Lee, and K. Ramchandran (2014). "The mds queue: analysing the latency performance of erasure codes". In: *IEEE International Symposium on Information Theory (ISIT '14)*, 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.

Sharma, S., S. Singh, and M. Sharma (2008). "Performance analysis of load balancing algorithms". *International Journal of Civil and Environmental Engineering* **2**:2, pp. 367–370.

Shivakumar, P. N. and K. H. Chew (1974). "A sufficient condition for nonvanishing of determinants". *Proceedings of the American Mathematical Society* **43**:1, pp. 63–66. ISSN: 00029939, 10886826. DOI: 10.2307/2039326.

Sideris, T. (2013). *Ordinary differential equations and dynamical systems*. Atlantis Press, Paris. ISBN: 978-94-6239-021-8.

Singh, S. and I. Chana (2015). "Qos-aware autonomic resource management in cloud computing: a systematic review". *ACM Comput. Surv.* **48**:3. ISSN: 0360-0300. DOI: 10.1145/2843889.

Spicuglia, S., L. Y. Chen, and W. Binder (2013). "Join the best queue: reducing performance variability in heterogeneous systems". In: *IEEE Sixth International Conference on Cloud Computing (CLOUD '13)*, pp. 139–146. DOI: 10.1109/CLOUD.2013.89.

Spinner, S., G. Casale, F. Brosig, and S. Kounev (2015). "Evaluating approaches to resource demand estimation". *Performance Evaluation* **92**, pp. 51–71. DOI: 10.1016/j.peva.2015.07.005.

Spinner, S., J. Grohmann, S. Eismann, and S. Kounev (2019). "Online model learning for self-aware computing infrastructures". *J. Systems and Software* **147**, pp. 1–16. DOI: 10.1016/j.jss.2018.09.089.

Tipper, D. and M. K. Sundareshan (1990). "Numerical methods for modeling computer networks under nonstationary conditions". *IEEE Journal on Selected Areas in Communications* **8**:9, pp. 1682–1695. DOI: 10.1109/49.62855.

Tribastone, M. (2013). "A fluid model for layered queueing networks". *IEEE Transactions on Software Engineering* **39**:6, pp. 744–756. DOI: 10.1109/TSE.2012.66.

Urgaonkar, B., G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi (2005). "An analytical model for multi-tier internet services and its applications". *SIGMETRICS Perform. Eval. Rev.* **33**:1, pp. 291–302. ISSN: 0163-5999. DOI: 10.1145/1071690.1064252.

Vaquero, L. M., L. Rodero-Merino, and R. Buyya (2011). "Dynamically scaling applications in the cloud". *SIGCOMM Comput. Commun. Rev.* **41**:1, pp. 45–52. ISSN: 0146-4833. DOI: 10.1145/1925861.1925869.

Varga, R. S. (2004). *Geršgorin and His Circles*. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-17798-9.

Vasantam, T., A. Mukhopadhyay, and R. R. Mazumdar (2018). "The mean-field behavior of processor sharing systems with general job lengths under the sq(d) policy". *Performance Evaluation* **127-128**, pp. 120–153. ISSN: 0166-5316. DOI: 10.1016/j.peva.2018.09.010.

Wang, H., J. Li, Z. Shen, and Y. Zhou (2018). "Approximations and bounds for (n, k) fork-join queues: a linear transformation approach". In: *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID '18)*. DOI: 10.1109/ccgrid.2018.00069.

Wang, Q., S. Zhang, Y. Kanemasa, C. Pu, B. Palanisamy, L. Harada, and M. Kawaba (2019). "Optimizing n-tier application scalability in the cloud: a study of soft resource allocation". **4**:2. ISSN: 2376-3639. DOI: 10.1145/3326120.

Wang, S., Y. Guo, N. Zhang, P. Yang, A. Zhou, and X. Shen (2021). "Delay-aware microservice coordination in mobile edge computing: a reinforcement learning approach". *IEEE Transactions on Mobile Computing* **20**:3, pp. 939–951. DOI: 10.1109/TMC.2019.2957804.

Wang, W. and G. Casale (2014). "Evaluating weighted round robin load balancing for cloud web services". In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 393–400. DOI: 10.1109/SYNASC.2014.59.

Wang, Y.-T. and Morris (1985). "Load sharing in distributed systems". *IEEE Transactions on Computers* **C-34**:3, pp. 204–217. DOI: 10.1109/TC.1985.1676564.

Wei-Ping Wang, D. Tipper, and S. Banerjee (1996). "A simple approximation for modeling nonstationary queues". In: *Proceedings of IEEE INFOCOM '96. Conference on Computer Communications*. Vol. 1, 255–262 vol.1. DOI: 10.1109/INFCOM.1996.497901.

Xiao, Y. and M. Krunz (2017). "Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pp. 1–9. DOI: 10.1109/INFOCOM.2017.8057196.

Xiong, P., Z. Wang, S. Malkowski, Q. Wang, D. Jayasinghe, and C. Pu (2011). "Economical and robust provisioning of n-tier cloud workloads: a multi-level control approach". In: *2011 31st International Conference on Distributed Computing Systems*, pp. 571–580. DOI: 10.1109/ICDCS.2011.88.

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. DOI: 10.1109/TPDS.2016.2564962.

Yanggratoke, R., J. Ahmed, J. Ardelius, C. Flinta, A. Johnsson, D. Gillblad, and R. Stadler (2015a). "Predicting service metrics for cluster-based services using real-time analytics". In: *2015 11th International Conference on Network and Service Management (CNSM '15)*, pp. 135–143. DOI: 10.1109/CNSM.2015.7367349.

Yanggratoke, R., G. Kreitz, M. Goldmann, R. Stadler, and V. Fodor (2015b). "On the performance of the spotify backend". *Journal of Network and Systems Management* **23**:1, pp. 210–237. DOI: 10.1007/s10922-013-9292-2.

Yao, Z., Y. Desmouceaux, J.-A. Cordero-Fuertes, M. Townsley, and T. Clausen (2022). "Hlb: toward load-aware load balancing". *IEEE/ACM Transactions on Networking*, pp. 1–16. DOI: 10.1109/TNET.2022.3177163.

Ying, L. (2017). "Stein's method for mean field approximations in light and heavy traffic regimes". *Proc. ACM Meas. Anal. Comput. Syst.* **1**:1. DOI: 10.1145/3084449.

Yousefpour, A., C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue (2019). "All one needs to know about fog computing and related edge computing paradigms: a complete survey". *Journal of Systems Architecture* **98**, pp. 289–330. DOI: 10.1016/j.sysarc.2019.02.009.

Yu, Y., J. Yang, C. Guo, H. Zheng, and J. He (2019). "Joint optimization of service request routing and instance placement in the microservice system". *Journal of Network and Computer Applications* **147**, p. 102441. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2019.102441.

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: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. USENIX Association, 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.

Zhu, L., G. Casale, and I. Perez (2020). "Fluid approximation of closed queueing networks with discriminatory processor sharing". *Performance Evaluation* **139**, p. 102094. ISSN: 0166-5316. DOI: 10.1016/j.peva.2020.102094.

A

Proofs from Chapter 3

This appendix contains proofs that were too long to smoothly include in the main text.

Proof of Theorem 3.1

First, notice that the drift for $(i, r, a) \in (\mathcal{Q}, \mathcal{C}_i, \mathcal{S}_{i,r})$ in (3.11) only depends on $\Psi^{i,r}$ in the first subexpression. Denote the remaining subexpressions as $v_{i,r,a}(\boldsymbol{X})$. Ordering the phases subsequently as described in Assumption 3.2, we can then express the drift over any $i \in \mathcal{Q},\ r \in \mathcal{C}_i$ as

$$F_{i,r,\cdot}(\boldsymbol{X}) = \left(\Psi^{i,r}\right)^T \theta_{i,r,\cdot}(\boldsymbol{X}) + v_{i,r,\cdot}(\boldsymbol{X}). \tag{A.1}$$

We can then introduce the block-diagonal matrix

$$\boldsymbol{\Psi}\left(\in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{S}|}\right) = \operatorname{diag}\left(\Psi^{1,1}, \Psi^{1,2}, \Psi^{1,3}, \ldots\right), \tag{A.2}$$

and get that

$$F(\boldsymbol{X}) = \boldsymbol{\Psi}^T \theta(\boldsymbol{X}) + v(\boldsymbol{X}). \tag{A.3}$$

For $v_{i,r,a}$, we start by moving constants out from the sums,

$$v_{i,r,a}(\boldsymbol{X}) = \zeta_a^{i,r} \sum_{j\in\mathcal{Q}} \sum_{s\in\mathcal{C}_j} P_{j,i}^{s,r} \sum_{b\in\mathcal{S}_{j,s}} \psi_b^{j,s} \theta_{j,s,b}\left(\boldsymbol{X}\right) + \zeta_a^{i,r} \lambda^{i,r}. \tag{A.4}$$

The final sum can be expressed as $\sum_{b\in\mathcal{S}_{j,s}} \psi_b^{j,s} \theta_{j,s,b}\left(\boldsymbol{X}\right) = \left(\psi^{j,s}\right)^T \theta_{j,s,\cdot}\left(\boldsymbol{X}\right)$. If we then introduce the block-diagonal matrix

$$\boldsymbol{B}\left(\in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}\right) = \operatorname{diag}\left(\psi^{1,1}, \psi^{1,2}, \psi^{1,3}, \ldots\right), \tag{A.5}$$

we get that

$$\left(\psi^{j,s}\right)^T \theta_{\vec{m}_{j,s}}\left(\boldsymbol{X}\right) = \left(\boldsymbol{B}^T \theta(\boldsymbol{X})\right)_{n_{j,s}}. \tag{A.6}$$

From Assumption 3.2, the entire routing probability matrix can be written as

$$\boldsymbol{P} = \begin{bmatrix} P^{\cdot,\cdot}_{1,1} & P^{\cdot,\cdot}_{1,2} & \cdots \\ P^{\cdot,\cdot}_{2,2} & P^{\cdot,\cdot}_{2,2} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}, \tag{A.7}$$

which gives that

$$\begin{aligned} v_{i,r,a}(\boldsymbol{X}) &= \zeta_a^{i,r} \sum_{j=1}^{\mathcal{Q}} \sum_{s=1}^{\mathcal{C}_j} P^{s,r}_{j,i} \left(\boldsymbol{B}^T \theta(\boldsymbol{X})\right)_{n_{j,s}} + \zeta_a^{i,r} \lambda^{i,r} \\ &= \zeta_a^{i,r} \left(\boldsymbol{P}^T \boldsymbol{B}^T \theta(\boldsymbol{X})\right)_{n_{j,s}} + \zeta_a^{i,r} \lambda^{i,r}. \end{aligned} \tag{A.8}$$

By introducing a third block-diagonal matrix,

$$\boldsymbol{A}\big(\in \mathbb{R}^{|\mathcal{S}|\times|\mathcal{C}|}\big) = \operatorname{diag}\big(\zeta^{1,1}, \zeta^{1,2}, \zeta^{1,3}, \ldots\big), \tag{A.9}$$

we end up with

$$v(\boldsymbol{X}) = \boldsymbol{A}\boldsymbol{P}^T \boldsymbol{B}^T \theta(\boldsymbol{X}) + \boldsymbol{A}\boldsymbol{\lambda}. \tag{A.10}$$

Finally, by combining (A.3) and (A.10), the drift of the entire population process is given by

$$\begin{aligned} F(\boldsymbol{X}) &= \boldsymbol{W}^T \theta(\boldsymbol{X}) + \boldsymbol{A}\boldsymbol{\lambda}, \\ \text{where} \quad \boldsymbol{W} &= \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T. \end{aligned} \tag{A.11}$$

Proof of Theorem 3.2

We will focus on the 1-norm, for which Lipschitz continuity of $F(\boldsymbol{x})$ in $\mathbb{R}_+^{|\mathcal{S}|\times 1}$ implies that

$$||F(\boldsymbol{x}) - F(\boldsymbol{y})||_1 = \left|\left|\boldsymbol{W}^T \theta(\boldsymbol{x}) - \boldsymbol{W}^T \theta(\boldsymbol{y})\right|\right|_1 \leq L \, ||\boldsymbol{x} - \boldsymbol{y}||_1 , \tag{A.12}$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}_+^{|\mathcal{S}|\times 1}$ and some constant L. As the induced 1-norm of a matrix is subordinant, breaking out $\boldsymbol{W}$ yields that

$$||F(\boldsymbol{x}) - F(\boldsymbol{y})||_1 \leq \left|\left|\boldsymbol{W}^T\right|\right|_1 \cdot ||\theta(\boldsymbol{x}) - \theta(\boldsymbol{y})||_1 . \tag{A.13}$$

As $\boldsymbol{W}$ is a constant matrix with finite elements, each element can be bounded with $\omega = \max_{i,j} |\boldsymbol{W}_{i,j}|$. Since the induced 1-norm of a matrix is defined as the maximum absolute column sum, i.e., $\max_{j\in 1..|\mathcal{S}|} \sum_i |\boldsymbol{W}_{i,j}|$, we get that

$$||F(\boldsymbol{x}) - F(\boldsymbol{y})||_1 \leq |\mathcal{S}|\omega \cdot ||\theta(\boldsymbol{x}) - \theta(\boldsymbol{y})||_1 . \tag{A.14}$$

Because of the minimums present in $\theta(\boldsymbol{x})$, we will consider each of the four cases separately. Let $i \in \mathcal{Q}$ and $\boldsymbol{x}_i$ correspond to the population vector of all phase states in queue i. Note that since each state belongs to exactly one queue, we can write $||\theta(\boldsymbol{x}) - \theta(\boldsymbol{y})||_1 = \sum_{i\in\mathcal{Q}} ||\theta_i(\boldsymbol{x}) - \theta_i(\boldsymbol{y})||_1$ where

$$||\theta_i(\boldsymbol{x}) - \theta_i(\boldsymbol{y})||_1 = \sum_j \left| x_{i,j} \frac{\min\left(k_i, \sum_j x_{i,j}\right)}{\sum_j x_{i,j}} - y_{i,j} \frac{\min\left(k_i, \sum_j y_{i,j}\right)}{\sum_j y_{i,j}} \right|. \tag{A.15}$$

From the four possible cases, three needs to be proven Lipschitz as two of the cases are identical due to symmetry. The three cases are

(i) $\sum_j x_{i,j} \leq k_i$, $\sum_j y_{i,j} \leq k_i$ which yields

$$\sum_j |x_{i,j} - y_{i,j}| = ||\boldsymbol{x}_i - \boldsymbol{y}_i||_1. \tag{A.16}$$

(ii) $\sum_j x_{i,j} \geq k_i$, $\sum_j y_{i,j} \leq k_i$ which yields

$$\begin{aligned} &\sum_j \left| x_{i,j} \frac{k_i}{\sum_j x_{i,j}} - y_{i,j} \right| \\ &\quad = \sum_j \left| \frac{k_i}{\sum_j x_{i,j}} (x_{i,j} - y_{i,j}) + y_{i,j} \left(\frac{k_i}{\sum_j x_{i,j}} - 1 \right) \right|. \end{aligned} \tag{A.17}$$

The triangle inequality together with $\sum_j x_{i,j} \geq k_i$ gives that

$$(\text{A.17}) \leq \frac{k_i}{\sum_j x_{i,j}} \sum_j |x_{i,j} - y_{i,j}| + \frac{\sum_j y_{i,j}}{\sum_j x_{i,j}} \left(\sum_j x_{i,j} - k_i \right). \tag{A.18}$$

Since $\sum_j y_{i,j} \leq k_i$ and $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}_+$ we get via the reverse triangle inequality that

$$\begin{aligned} (\text{A.18}) &\leq \frac{k_i}{\sum_j x_{i,j}} \sum_j |x_{i,j} - y_{i,j}| + \frac{\sum_j y_{i,j}}{\sum_j x_{i,j}} \sum_j |x_{i,j} - y_{i,j}| \\ &\leq 2||\boldsymbol{x}_i - \boldsymbol{y}_i||_1. \end{aligned} \tag{A.19}$$

(iii) $\sum_j x_{i,j} \geq k_i$, $\sum_j y_{i,j} \geq k_i$ which yields

$$\sum_j \left| x_{i,j} \frac{k_i}{\sum_j x_{i,j}} - y_{i,j} \frac{k_i}{\sum_j y_{i,j}} \right| = k_i \sum_j \left| \frac{1}{\sum_j x_{i,j}} (x_{i,j} - y_{i,j}) + y_{i,j} \left(\frac{1}{\sum_j x_{i,j}} - \frac{1}{\sum_j y_{i,j}} \right) \right|. \tag{A.20}$$

Using the triangle inequality

$$(\text{A.20}) \leq \frac{k_i}{\sum_j x_{i,j}} \sum_j |x_{i,j} - y_{i,j}| + k_i \sum_j y_{i,j} \left| \frac{1}{\sum_j x_{i,j}} - \frac{1}{\sum_j y_{i,j}} \right|. \tag{A.21}$$

Breaking out the denominators gives

$$(\text{A.21}) = \frac{k_i}{\sum_j x_{i,j}} \sum_j |x_{i,j} - y_{i,j}| + \frac{k_i}{\sum_j x_{i,j}} \left| \sum_j y_{i,j} - \sum_j x_{i,j} \right|. \tag{A.22}$$

Then the reverse triangle inequality together with $\sum_j x_{i,j} \geq k_i$ yields that

$$(\text{A.22}) \leq 2||\boldsymbol{x}_i - \boldsymbol{y}_i||_1. \tag{A.23}$$

Hence,

$$||\theta(\boldsymbol{x}) - \theta(\boldsymbol{y})||_1 = \sum_{i \in \mathcal{Q}} ||\theta_i(\boldsymbol{x}) - \theta_i(\boldsymbol{y})||_1 \leq 2||\boldsymbol{x} - \boldsymbol{y}||_1, \tag{A.24}$$

and $||F(\boldsymbol{x}) - F(\boldsymbol{y})||_1 \leq 2|\mathcal{S}|\omega\, ||\boldsymbol{x} - \boldsymbol{y}||_1$.

B

Running Example from Chapter 3

This appendix gives a detailed explanation on how to adapt some of the methods in Chapter 3 for the second running example.

Compact Matrix-Form for Example 2

For our closed cyclic network, the routing probability matrix becomes

$$\boldsymbol{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \tag{B.1}$$

and since we have no external arrivals due to the network being closed, $\boldsymbol{\lambda} = \mathbf{0}$. Stacking the PH matrices in block diagonals yields

$$\boldsymbol{\Psi} = \begin{bmatrix} -\mu_1 & 0 & 0 & 0 & 0 \\ 0 & -4.0 & 4.0 & 0 & 0 \\ 0 & 0 & -4.0 & 0 & 0 \\ 0 & 0 & 0 & -2.0 & 0.1 \\ 0 & 0 & 0 & 0 & -0.1 \end{bmatrix},$$
$$\boldsymbol{B} = \begin{bmatrix} \mu_1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 4.0 & 0 \\ 0 & 0 & 1.9 \\ 0 & 0 & 0.1 \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{B.2}$$

which gives that

$$\boldsymbol{W} = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}\boldsymbol{A}^T = \begin{bmatrix} -\mu_1 & \mu_1 & 0 & 0 & 0 \\ 0 & -4.0 & 4.0 & 0 & 0 \\ 0 & 0 & -4.0 & 4.0 & 0 \\ 1.9 & 0 & 0 & -2.0 & 0.1 \\ 0.1 & 0 & 0 & 0 & -0.1 \end{bmatrix},$$

$$\theta(\boldsymbol{x}) = \begin{bmatrix} x_1 \\ x_2 \cdot \min(4, x_2 + x_3)/(x_2 + x_3) \\ x_3 \cdot \min(4, x_2 + x_3)/(x_2 + x_3) \\ x_4 \cdot \min(8, x_4 + x_5)/(x_4 + x_5) \\ x_5 \cdot \min(8, x_4 + x_5)/(x_4 + x_5) \end{bmatrix}, \tag{B.3}$$

from which the mean-field fluid model is obtained.

Response Time CDF Approximation for Example 2

In order to employ the approximation (3.48), we must first find values on $\boldsymbol{W}_R$, $\hat{g}(\boldsymbol{\eta})$ and $\hat{\beta}(\boldsymbol{\eta})$.

First, only considering the transitions between classes in $\mathcal{C}_R$ yields that

$$\boldsymbol{P}_R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \tag{B.4}$$

which in turn gives that

$$\boldsymbol{W}_R = \boldsymbol{\Psi} + \boldsymbol{B}\boldsymbol{P}_R\boldsymbol{A}^T = \begin{bmatrix} -\mu_1 & 0 & 0 & 0 & 0 \\ 0 & -4.0 & 4.0 & 0 & 0 \\ 0 & 0 & -4.0 & 4.0 & 0 \\ 0 & 0 & 0 & -2.0 & 0.1 \\ 0 & 0 & 0 & 0 & -0.1 \end{bmatrix}. \tag{B.5}$$

At $\mu_1 = 0.2$ and $\hat{sv}_2 \approx 2.24, \hat{\eta}_3 \approx 3.5$, the smoothed mean-field fluid model gives the fixed point

$$\boldsymbol{x}^*(\hat{\boldsymbol{\eta}}) \approx \begin{bmatrix} 34.6 & 3.2 & 3.2 & 4.5 & 4.5 \end{bmatrix}^T, \tag{B.6}$$

and thus the processor share approximation becomes

$$\hat{g}(\hat{\boldsymbol{\eta}}) \approx \begin{bmatrix} 1 & 0.57 & 0.57 & 0.77 & 0.77 \end{bmatrix}^T. \tag{B.7}$$

The final piece is to calculate the probability that a request arrives to a specific class in $\mathcal{C}_R$ at stationarity. First, the inflow connections from outside

$\mathcal{C}_R$ becomes $\boldsymbol{P}_{\lambda\left(\mathcal{C}_R^c\right)} = \begin{bmatrix}1 & 0\end{bmatrix}$. Further, the approximation to the average outflow across all classes in $\mathcal{C}$ becomes

$$\begin{aligned}\hat{\boldsymbol{\mu}}_d\left(\hat{\boldsymbol{\eta}}\right) &= \boldsymbol{B}^T D^{\hat{g}\left(\hat{\boldsymbol{\eta}}^*\right)} \boldsymbol{x}^*\left(\hat{\boldsymbol{\eta}}^*\right) \\ &= \begin{bmatrix}0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4.0 & 0 & 0 \\ 0 & 0 & 0 & 1.9 & 0.1\end{bmatrix} \begin{bmatrix}1.0 & 0 & 0 & 0 & 0 \\ 0 & 0.57 & 0 & 0 & 0 \\ 0 & 0 & 0.57 & 0 & 0 \\ 0 & 0 & 0 & 0.77 & 0 \\ 0 & 0 & 0 & 0 & 0.77\end{bmatrix} \\ &\cdot \begin{bmatrix}34.6 \\ 3.2 \\ 3.2 \\ 4.5 \\ 4.5\end{bmatrix} = \begin{bmatrix}6.92 \\ 7.3 \\ 6.92\end{bmatrix}. \end{aligned} \tag{B.8}$$

Hence

$$\hat{\beta}^v_{\mathcal{C}_R}\left(\hat{\boldsymbol{\eta}}^*\right) = \boldsymbol{P}^T_{\lambda\left(\mathcal{C}_R^c\right)} \left[\hat{\boldsymbol{\mu}}_d\left(\hat{\boldsymbol{\eta}}^*\right)\right]_{\mathcal{C}_R^c} = \begin{bmatrix}6.92 \\ 0\end{bmatrix}. \tag{B.9}$$

and thus we get the probability of request arrival as $\hat{\beta} = \begin{bmatrix}0 & 1 & 0\end{bmatrix}^T$, resulting in the following response time CDF approximation over $\mathcal{C}_R$ at stationarity with $\mu_1 = 0.2$:

$$\Theta_{\mathcal{C}_R}\left(t \mid \hat{\boldsymbol{\eta}}^*\right) = 1 - \begin{bmatrix}0 & 1.0 & 0 & 0 & 0\end{bmatrix} \tag{B.10}$$

$$\cdot \exp\left(\begin{bmatrix}-0.2 & 0 & 0 & 0 & 0 \\ 0 & -2.28 & 2.28 & 0 & 0 \\ 0 & 0 & -2.28 & 0 & 0 \\ 0 & 0 & 0 & -1.54 & 0.077 \\ 0 & 0 & 0 & 0 & -0.077\end{bmatrix} \cdot t\right) \mathbb{1}. \tag{B.11}$$

This however only holds for the current system, when some perturbation is introduced by, e.g., changing μ_1, the values for $\boldsymbol{x}^*\left(\boldsymbol{\eta}\right)$, $\hat{g}\left(\boldsymbol{\eta}\right)$ and $\hat{\beta}\left(\boldsymbol{\eta}\right)$ need to be re-calculated.