



LUND UNIVERSITY

Control over the Cloud

Offloading, Elastic Computing, and Predictive Control

Skarin, Per

2021

Document Version:

Publisher's PDF, also known as Version of record

[Link to publication](#)

Citation for published version (APA):

Skarin, P. (2021). *Control over the Cloud: Offloading, Elastic Computing, and Predictive Control*. Department of Automatic Control, Lund University.

Total number of authors:

1

General rights

Unless other specific re-use rights are stated the following general rights apply:

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

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

Take down policy

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

LUND UNIVERSITY

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



Control over the Cloud

Offloading, Elastic Computing,
and Predictive Control

PER SKARIN

DEPARTMENT OF AUTOMATIC CONTROL | LUND UNIVERSITY

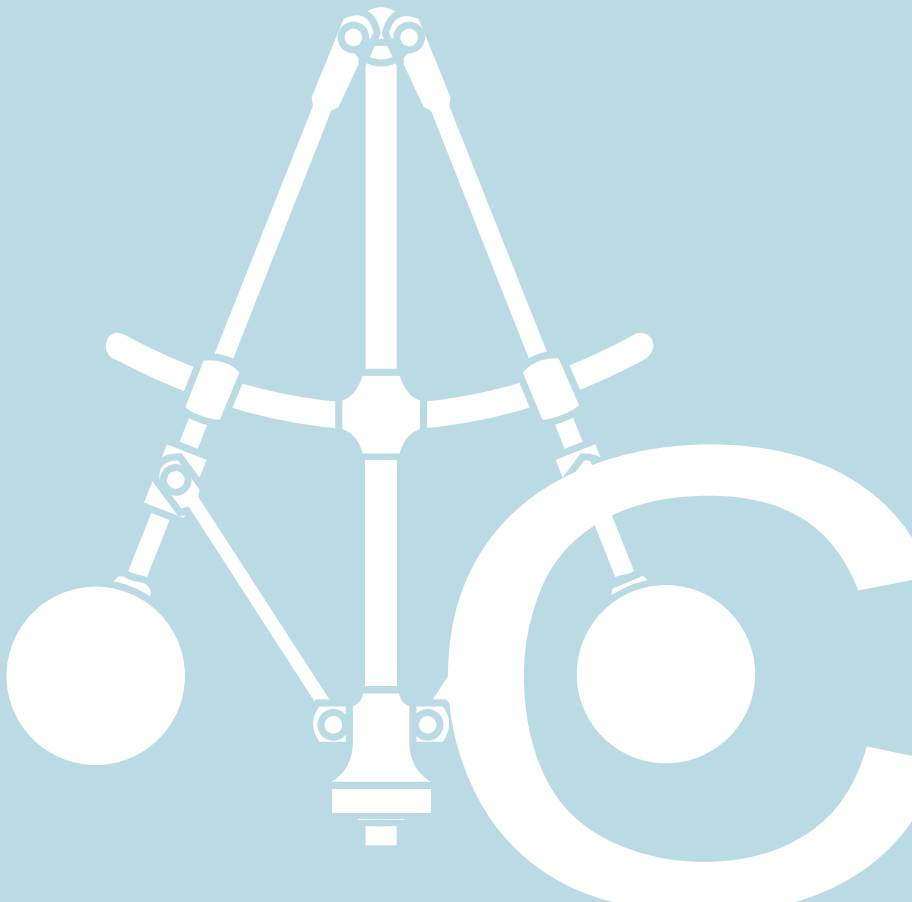




LUND
UNIVERSITY

Department of Automatic Control
P.O. Box 118, 221 00 Lund, Sweden
www.control.lth.se

PhD Thesis TFRT-1132
ISBN 978-91-8039-093-4
ISSN 0280-5316



Control over the Cloud

Offloading, Elastic Computing,
and Predictive Control

Per Skarin



LUND
UNIVERSITY

Department of Automatic Control

Cover Illustration: The cover photo shows the inside of a data center. The image is courtesy of Ericsson AB.

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

Ph.D. Thesis TFRT-1132
ISBN 978-91-8039-093-4 (print)
ISBN 978-91-8039-094-1 (web)
ISSN 0280-5316

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

© 2021 by Per Skarin. All rights reserved.
Printed in Sweden by Media-Tryck.
Lund 2021

Abstract

The thesis studies the use of cloud native software and platforms to implement critical closed loop control. It considers technologies that provide low latency and reliable wireless communication, in terms of edge clouds and massive MIMO, but also approaches industrial IoT and the services of a distributed cloud, as an extension of commercial-of-the-shelf software and systems.

First, the thesis defines the cloud control challenge, as control over the cloud and controller offloading. This is followed by a demonstration of closed loop control, using MPC, running on a testbed representing the distributed cloud. The testbed is implemented using an IoT device, clouds, next generation wireless technology, and a distributed execution platform. Platform details are provided and feasibility of the approach is shown. Evaluation includes relocating an on-line MPC to various locations in the distributed cloud.

Offloaded control is examined next, through further evaluation of cloud native software and frameworks. This is followed by three controller designs, tailored for use with the cloud. The first controller solves MPC problems in parallel, to implement a variable horizon controller. The second is a hierarchical design, in which rate switching is used to implement constrained control, with a local and a remote mode. The third design focuses on reliability. Here, the MPC problem is extended to include recovery paths that represent a fallback mode. This is used by a control client if it experiences connectivity issues. An implementation is detailed and examined.

In the final part of the thesis, the focus is on latency and congestion. A cloud control client can experience long and variable delays, from network and computations, and used services can become overloaded. These problems are approached by using predicted control inputs, dynamically adjusting the control frequency, and using horizontal scaling of the cloud service. Several examples are shown through simulation and on real clouds, including admitting control clients into a cluster that becomes temporarily overloaded.

Acknowledgments

I'd first like to acknowledge and thank Johan Eker, who got me into this work. Johan is a great person to be around, professionally and personally. After meeting Johan once in 2005, I was intrigued by the idea of someday making my way into Ericsson Research. Ten years later, I found myself joining Johan's team. A shared interest in automatic control soon also led to WASP.

Supervisors Karl-Erik Årzén and Maria Kihl have been instrumental throughout these years. Without their support, professionalism, and faith, I would have been hard pressed to condense my vision and set of ideas into focused papers, and this thesis. The Ph.D. work is a learning process, and Karl-Erik and Maria are great teachers, sometimes tough, always fair, and at all times in a good mood. Special thanks to Karl-Erik who despite many other duties has invested a lot of time and effort.

Several of the experiments, thought processes, and papers inside this thesis were created jointly with William Tärneberg. William is also the artist behind many great visualizations that exist around this work (Figures 1.3, 2.4 and 8.1 in the thesis are courtesy of William). Thank you William for making research such a joy, and for your kindness, perseverance, and great work - and for challenging *me* technically and intellectually. Thanks also to Fredrik Tufvesson and others at the EIT department for providing support early in this project and to Tarek Abdelzaher for the cloud control challenge.

Many thanks to The Knut and Alice Wallenberg Foundation for this opportunity, to Ericsson AB and to Joakim Persson for supporting my application to WASP. Thanks also to all supportive colleagues at Ericsson and Ola Angelsmark for providing me peace of mind to finish the thesis. The ELLIIT strategic research area on IT and mobile communications and the Nordforsk university hub on Industrial IoT (HI2OT) have also been supportive in this work. Thanks to Pontus Giselsson and Leif Andersson for helping me finalize the thesis and thanks to everyone else at the automatic control department in Lund - you truly make it a great place to work and a great place to be.

I would also like to extend my gratitude to all of the supportive, knowledgeable, and inspirational people that I have met through WASP. A spe-

cial thanks to everyone in the first batch of WASP-AS students who made it a great pleasure to attend courses, conferences, and travels. And to Bo Bernardsson, Fredrik Heintz, Christian Berger, and Patric Jensfelt of the WASP graduate school for all their arrangements with courses and study trips, and for bringing their great personalities to these events. Special thanks also to the people working with the WARAs, directors Fredrik Dahlberg and Gunnar Bark, Jesper Tordenlid, Fredrik von Corswant, fellow Ph.D. students, and to the people at Saab, Combitech, Axis, SMaRC and the Swedish Sea Rescue Society. Related to this research, I've had the pleasure of working with the WARAs in my roles as a Ph.D. student, mediator, and engineer. It has been inspiring and always a pleasure. Thanks to Erik Elmroth and his team in Umeå for arranging the Cloud Control Workshops, and other arrangements for the Autonomous Cloud projects within WASP.

A special mention to M. Alexandre Martin with whom I had the pleasure of sharing an office for three years, and who arranged and organized two ample research visits to Japan and Singapore. Alors je te remercie d'être un collègue excellent et un bon ami aussi. Thanks also to Amir Roozbeh, Joris van Rooij, Piergiuseppe Mallozzi, and Rebekka Wohlrab for the great company on those trips, discussing research and creating memories for life.

Finally, thanks to my family and to my extended family. To my talented, ambitions siblings, who are always pushing me to go further. To my father, who inspired my path to computer engineering and subsequently inspired me to work with automatic control. To my mother, who unfortunately left us. To Jessica, who I owe my life and character. Thank you for always being there and for keeping me focused. And to my wonderful daughters Elsa and Clara. As I am writing this, Elsa is reading a bedtime story to Clara and your voices sneak into my study. It is sentimental to me that you have endured a long and isolating pandemic, and towards the end of it, a father hard at work. As both of these things are now coming to an end, I long for the ordinary days spent with you. Nothing is more rewarding than that.

Per Skarin

Lund, November 2021

Funding

Ericsson AB has partly funded this works as an industrial Ph.D. project. This work was also partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

Contents

1. Introduction	11
1.1 Motivation	15
1.2 Scope of the thesis	18
1.3 Thesis outline and contributions	19
1.4 Publications	21
1.5 Notation	25
Glossary	27
Acronyms	29
Part I Cloud Control Challenge	33
2. Intelligence in the Cloud	35
2.1 Cloud Overview	36
2.2 Contemporary Cloud	40
2.3 Research Challenges	44
2.4 Controller offloading	45
2.5 Closed loop guarantees	49
3. Control over the Cloud	50
3.1 Networked control	50
3.2 Real-time	53
3.3 Cloud control system	54
3.4 Control over the Cloud	55
4. Model Predictive Control	58
4.1 Definition	59
4.2 Stability	61
4.3 The basic MPC - limitations and extensions	65
4.4 Linear and quadratic control	66
5. Reference Plant	71

Part II Cloud Performance	75
6. Introduction	77
6.1 Related work	78
6.2 Research gap	80
7. Towards Mission Critical Control	82
7.1 5G enabled testbed	82
7.2 Edge cloud and massive MIMO	83
7.3 Cloud native application	87
7.4 Implementation	90
7.5 Testbed evaluation	91
7.6 Conclusions	97
8. Function Service Performance	98
8.1 Objective	99
8.2 Platform	100
8.3 Experiments	101
8.4 Computational load	103
8.5 Cloud stack latency progression	105
8.6 Feedback control	113
8.7 Conclusions	121
Part III Predictive Control in the Cloud	123
9. Introduction	125
9.1 Related Work	126
9.2 Research Gap	128
9.3 General Design	129
10. Variable Horizon Control	132
10.1 Targeted system	132
10.2 Controller	133
10.3 Evaluation	140
10.4 Edge perspective	147
10.5 Conclusion	151
11. Rate Switching	152
11.1 Targeted system	152
11.2 Controller	154
11.3 Evaluation	159
11.4 Conclusions	167
12. Explicit Recovery	168
12.1 Setup	168
12.2 Controller	170
12.3 Evaluation	178

12.4 Conclusion	184
Part IV Adaptation	185
13. Resilient Elastic Control	187
13.1 Background and related work	188
13.2 Simulations	191
13.3 Offloaded controller	195
14. Resilient Elastic Control in the Cloud	207
14.1 Frequency Controller Details	208
14.2 Experimental setup	213
14.3 Results	215
14.4 Conclusions	222
Part V Conclusions	225
15. Conclusions and Future Work	227
15.1 Future work	229
Bibliography	232
A. Cost tables	246

1

Introduction

The internet and worldwide web initiated a communications revolution which has changed the world. [...] It is an educated guess that it will also have a very strong impact on automatic control.

[Åström and Kumar, 2013]

In the 1960s, the advent of digital computers started a golden age for automatic control. In the decades that followed, computing and communication systems had a tremendous impact on advancing technology and automation, and subsequently on the necessary control engineering [Åström and Kumar, 2013]. In the early information age, digital electronics, servo systems and negative feedback were important parts in shaping industry and technology to form the advanced systems on which we now largely depend.

The focus shifted at the turn of the century with the arrival of the World Wide Web (WWW). The new form of interaction that the WWW brought had a profound, and different, impact on society. Software companies became the technology drivers as Internet services and the end user applications become more important. This shift marks the second part of the information age, driven by the subsequent revolution in digital communications. While the Internet and the WWW are well established as drivers behind this part of our history, in today's retrospect, the parallel evolution of cellular communications is perhaps as important. At present, these two technological drivers, wireless communication and Internet technology, have evolved to a state where they are again becoming disruptive as they enter into industrial automation. The merger of these two paths of history are the reasons behind this thesis, and we therefore start with a short perspective on the lineages of the Internet and of cellular technology.

A brief communication history

A few key events in the evolution of the Internet and cellular broadband are shown in Figure 1.1. The first two events are the Advanced Research Projects Agency Network (ARPANET) and the AXE. ARPANET, a precursor to the Internet, was the world's first decentralized, packet switched network, developed by the United States military. It is often marked as the start of the Internet. AXE was a circuit switched, modularized, digital telephone exchange developed by Ericsson and the Swedish public enterprise Televerket (now Telia AB). It was developed for landline communication but would become a core component also in the early cellular systems. The ARPANET and AXE are essential in this history because of their significance as starting points for the two global communication networks, but the trajectories of the Internet and telecommunication, as we know them today, really start in the early 1980s.

In 1981, Nordic Mobile Telephony (NMT) was launched as the first international cellular network for mobile telephony. Shortly thereafter, the worldwide packet-based communication standard of TCP/IP was deployed, a technology that is ubiquitous in today's communication systems. The WWW was released in 1991, the same year as the second generation of cellular networks, Global System for Mobile Communications (GSM). In this generation, phones sent text messages and had some capacity for data traffic but did not extensively communicate with the Internet. In the GSM era, the focus was on quality voice calls and text messages, and in the later stage, limited access to the WWW through the Wireless Application Protocol (WAP). In 2001, 3G, the third generation mobile telephony, shifted focus from the circuit switched, quality of service voice calls, to shared high speed data channels. The cellular systems now focused on transferring much more data and on accessing digital services. Then, in 2006 cloud computing, which underlies the work in this thesis, became an emerging technology [Zakon, 2018]. Around the same time, the smartphone was introduced on the market. The telephone in everyone's pocket, now started to interact directly with the same Internet infrastructure that was accessed by stationary computers.

Cloud technology, sprung from the Internet ecosystem, and the smartphones, from the cellular domain, began to shape what is sometimes referred to as the app economy [Godfrey et al., 2016]. The Internet-of-things (IoT), listed as a disruptive civil technology by the US National Intelligence Council in 2008 [Atzori et al., 2010], also became a popular term. It would take a few years, but in 2013 IoT really started trending globally [Google, 2021], and the term has now made its way into industry through Industry 4.0 and Industrial IoT. Meanwhile, cloud has evolved from delivering social networks and web shops, to an ecosystem of developer tools, and an indispensable part of society. Communication technology would come to be shaped by smart-

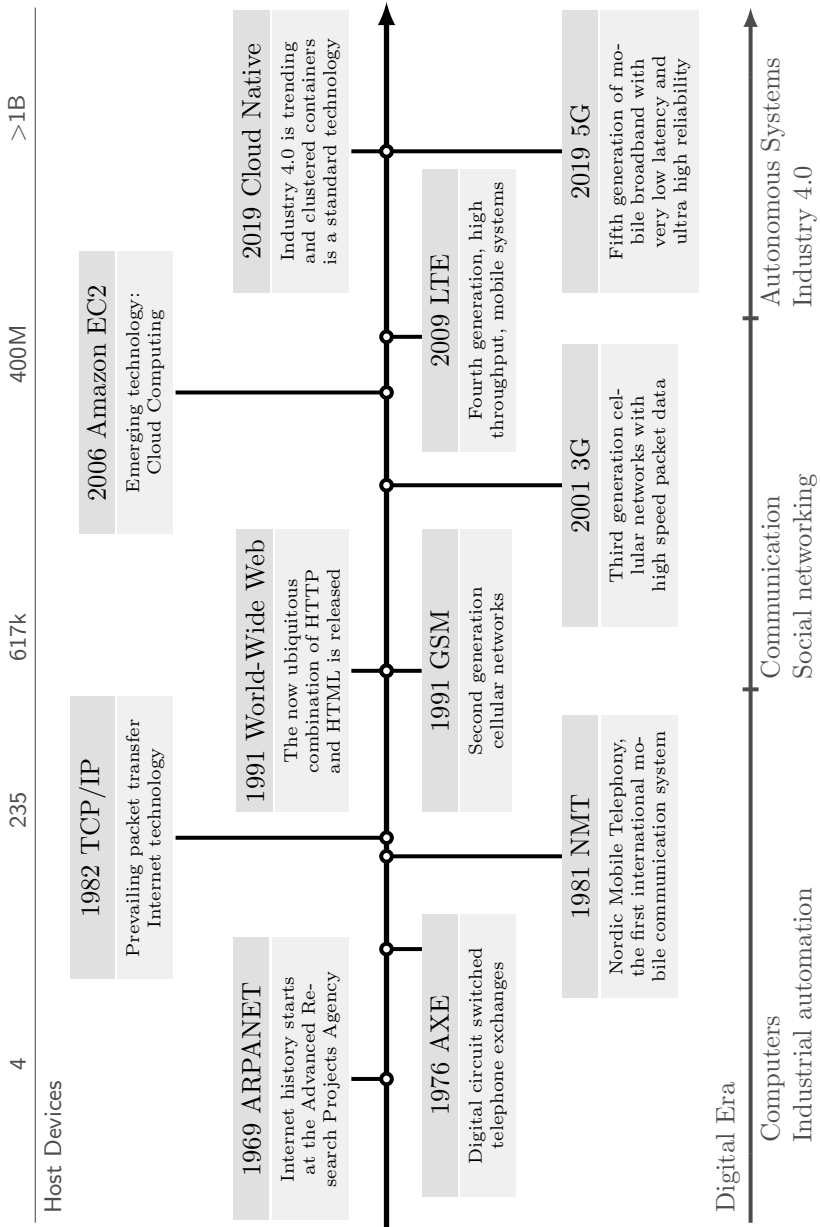


Figure 1.1 Internet and communication technology evolution. [Zakon, 2018; McCarthy, 2001; Science and Media Museum, 2020; Internet Systems Consortium, 2019; Ohlsson et al., 2015]

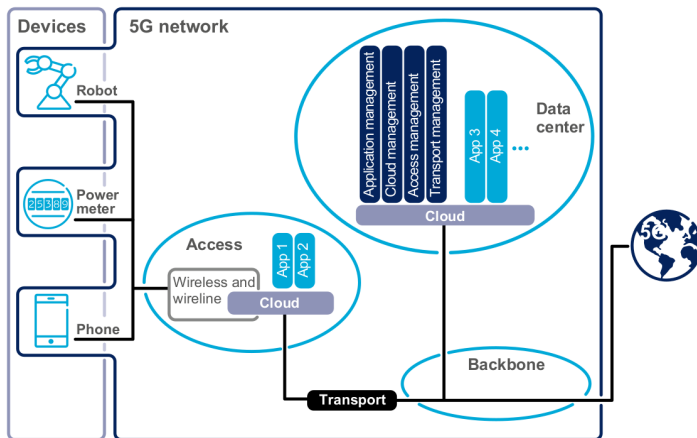


Figure 1.2 The 5G system with two clouds integrated into the core network. One cloud is integrated close to the access network, providing low latency. Image courtesy of Ericsson. Source: [Ericsson, 2017]

phones and other equipment needing access to remote services through the communication networks. Now, with the advent of the 5th generation of mobile telephony, 5G, cellular networks are not only an access point to cloud services, it is actively promoting, providing and using the technology, as illustrated in Figure 1.2. In relation to these developments, there has also been a new surge in machine learning, a field that is now providing powerful input to the engineering toolbox. While this thesis does not deal with machine learning directly, it is an important part of the perspective.

Autonomous systems

In combination, the maturity level of these breakthroughs, brings us to a new age in the digital era. It marks the entry point of disruptive Internet technologies and related commercial off-the-shelf (COTS) components in industry, and a revival of massive automation, using the new technologies. In Figure 1.1, this is referred to as the age of autonomous systems. Smart manufacturing and cloud robotics are part of this digital revolution, as are autonomous transports, the digital assistant, health services and many other developments driven by availability, devices, digitization, machine learning and communication networks. This progress was identified by the Knut and Alice Wallenberg Foundation which launched the Wallenberg Autonomous Systems Program (WASP) in 2015. It was soon to become the Wallenberg AI, Autonomous Systems and Software Program, making explicit two important components (i.e. software and AI) in the automation of complex systems. These systems are meant to observe, interact with, and adapt to the world.

When the graduate school of WASP was launched, automatic control had ample presence, as had the cloud, promising to manage massive amounts of information and fulfill every computational need. At this point in time, cloud was no longer an emerging technology but the public cloud ecosystem was still settling. Some researchers in automatic control had taken a specific interest in using their tools and knowledge to improve these large dynamic systems. Largely though, the goal in the next generation of clouds would be to support applications that are time sensitive and business/safety critical, and must respond to stimuli in the fraction of a second. Using the cloud for such applications is the topic of this thesis.

1.1 Motivation

The motivation behind this work is the evolving ecosystem around cloud, IoT, and wireless communication. Technology has reached a point where these systems can achieve a performance and reliability that allows them to co-exist with industrial grade equipment. With a bit error rate of 1×10^{-5} in combination with sub-millisecond delay in the access network, the Fifth Generation Wireless Specifications (5G) will provide mobile broadband with ultra-reliable and low-latency communication. The 5G system will also provide flexible allocation of services within the network. This opens up for wireless, latency sensitive, critical, closed-loop control systems to move into and draw benefit from cloud technologies. 5G and cloud services are enablers for new applications and industries that will evolve in what is known as the fourth industrial revolution.

A programmable logic controller (PLC) that is implemented in the cloud, is referred to as a virtual PLC. A virtual PLC, which forms a direct replacement of its physical counterpart, can provide simplified development, lower up-front cost, a faster time to market, can easily be replicated, replaced, extended, and monitored, and is provided fast access to a huge database of information. COTS, and IoT, allows fast and easy deployment of massive amounts of low-power devices. Complex systems such as autonomous personal transports, cloud robotics, and drones are also connected to the infrastructure and will benefit from using it efficiently. Industry 4.0 envisions smart factories and new industries empowered by the flexibility of this new ecosystem and the synergies of devices, clouds and low latency communication. The next generation communication networks are built to provide features that can make this vision a reality. Before setting the scope of the thesis, the remainder of this chapter briefly introduces central topics on this theme.

Cloud computing

The term cloud computing was popularized in the first decade of the 21st century at the arrival of public pay-as-you-go large-scale data center service offerings. This is often marked by the relaunch of Amazon Web Services in 2006. In this context, the term cloud soon became synonymous with the use of Internet based web services to store data and access applications. The e-commerce, social networks, and on-demand media providers that support much of modern lifestyle, depend on technologies that emerged in the cloud. The technologies support scalable applications executing in data centers.

Clouds provide *utility computing*, the meaning and importance of which is best described by the quote introducing Chapter 2 of this thesis, a quote from computer pioneer John McCarthy, in 1961. At the time of McCarthy's speech, architects at International Business Machines (IBM) were modifying their designs to support time-sharing on main-frame computers used inside large organizations. Today, this sharing happens on a massive scale, as a large part of the computational needs are serviced through hyperscale data centers with a shared infrastructure available publicly over the Internet. The cloud has come to represent the availability of computing infrastructure and services in resemblance with a public utility such as electricity. For many, it is an ubiquitous everyday necessity, accessible at all times.

This ecosystem of web services, storage, and compute resources, that are made available over remote networks and paid per usage, gained momentum roughly one decade ago ([Armbrust et al., 2010]). Since then, cloud has revolutionized the software industry and is increasingly becoming an intrinsic part of our infrastructure. The environment created by cloud, and the massive amounts of data passing through them, has also paved the way for the surge of interest in machine learning and artificial intelligence, cloud robotics, extensive monitoring and fault detection etc. Another indicator of success, is its presence in information and communication systems in general, marked by how 5G mobile broadband systems are being implemented and perceived. Consequently, clouds are developing to support new use cases.

Distributed Cloud

Recent advances in cloud technology include concepts such as edge, fog, distributed clouds, osmotic computing, and the network compute fabric ([Yousefpour et al., 2019; Boberg et al., 2018; Villari et al., 2016; Sefidcon et al., 2021]). Edge computing brings computations and storage closer to the data sources to provide data locality, low latency and high reliability. The fog aims to bridge IoT and cloud in a continuum, including intermediate network services. Osmotic computing focus on the migration of *micro services* across data centers to the network edge. With a combination of rich sensor environments and the edge, cloud technology is gaining traction

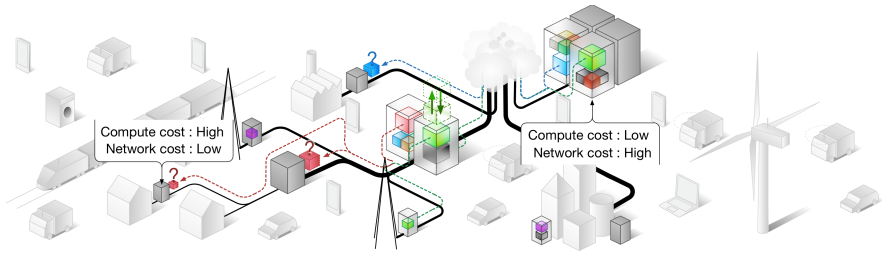


Figure 1.3 Distributed cloud. The edge node provides low latency, but has limited capacity for computations and storage. The data center has virtually infinite resources, but network access is slower and less reliable.

in traditional industries including logistics, transport, factory automation, manufacturing, and even for critical closed loop control systems. To simplify this process, the distributed cloud provides the necessary connectivity and the flexibility of cloud computing, while it can hide the complexity of the necessary infrastructure. An illustration of the distributed cloud is shown in Figure 1.3. Various levels in the infrastructure can service applications with different latency needs. Performing calculations close to the user can also off-load the communication paths in the network. The distributed cloud places software components in optimal locations to utilize the characteristics of the communication network and the serviced applications.

The fourth industrial revolution

Current technological advancement impacts the industrial taxonomy and allows industries to modernize and evolve. Industry 4.0 [Rojko, 2017; Khan et al., 2017; Jasperneite et al., 2020] is an originally German strategic initiative which has become synonymous with this revolution¹. Industry 4.0 revolves around cyber-physical system (CPS) (physical systems integrated with information and communication technology (ICT) components), industrial Internet-of-things (IIoT), and the smart factory. The machines in Industry 4.0 are autonomous systems, often mobile, capable of self-organization and self-optimization. The manufacturing system in a smart factory is re-configurable and processes should be adaptive, robust and user-friendly. Interoperability and wireless connectivity are important in Industry 4.0.

As the trend of connecting things continues, there is an understanding that responsiveness and reliability are set to improve significantly, through technologies such as distributed clouds, Time Sensitive Networks (TSN) and 5G connectivity. A complementary challenge to control theory and computer

¹There are several related and similar ideas around the world, such as the Industrial Internet and Industrie du futur

engineering is to identify useful trade-offs between latency and reliability, and the gains from using cloud technology. In the spirit of the fourth industrial revolution and Industry 4.0, control applications should provide self-organizing and self-optimizing systems, capable of making trade-offs online, through an awareness about their deployment.

Commercial off-the-shelf components

To realize the cloud-based systems aimed for the Industry 4.0 era, at reasonable costs and development effort, a design criterion is that they rely on tried and trusted commercial off-the-shelf (COTS) components. The open reference architecture for fog computing specifically requires that nodes in a fog network support COTS software [OpenFog Consortium, 2017]. This also relates to the interoperability requirements of Industry 4.0. Today, cloud-native is the culmination of COTS, cloud, and no-ops² software development and deployment. It is therefore reasonable to consider COTS components, services, and platforms as referring to, for example, public and private cloud providers (e.g. Amazon Web Services (AWS) and Microsoft Azure), open-source orchestration and service abstraction platforms (e.g. Kubernetes (K8S) [Bernstein, 2014], AWS Lambda [Fox et al., 2017], Fission [Mohanty et al., 2018], Robot Operating System (ROS) [Quigley et al., 2009]) and network services, in addition to traditional software. The fact that COTS is expected in relation to Industry 4.0, IoT, and cloud, is important, but the idea of COTS in control systems is not new or controversial [Árzén, 1999; Sha, 2001; Cervin et al., 2003; Kim et al., 2006].

1.2 Scope of the thesis

In the broader perspective, this thesis is about control engineering in the context of distributed clouds. The focus is on investigating control offloading through empirical evaluation of clouds and investigations into the use of utility computing. The cyber-physical systems, smart factories, multi-agent systems, and general *smart systems* that are the targets of this work will be highly advanced and complex. The thesis nonetheless focuses on a single control system using the cloud. There are good reasons for this, and four are listed below.

- 1) The field of control engineering lacks work that takes a specific interest in basic achievable performance in relation to conventional clouds.

²Refers to the automation of the IT environment removing the need for an organization to include an operations team to manage it.

- 2) Starting from conventional control of a single plant will help in identifying primitives that set cloud control apart from traditional control and networked control.
- 3) It is common to reject the use of cloud for critical control because of uncertainty or, write off its properties to consider it as just another networked control system. A use case that is well understood and easy to implement and measure, is a good way to challenge the current view on the cloud.
- 4) The use of elastic computing in complex critical systems will be easier to adopt (by reducing risk) if the individual parts are created to be resilient; tolerating variations and working at reduced performance if necessary.

Access networks will become reliable, and developers may be provided support for deterministic execution, but clouds are large and complex dynamic systems that are shared per definition, with resources acquired on demand. It is reasonable to assume that determinism cannot always be granted. It is also conceivable that determinism is not always preferred, if the alternative is an improved average performance. Therefore, the aim of the thesis is to specifically examine clouds and utility computing. While experiments are performed over networks, subjected to network reliability and delays, a focus on traditional networked control is avoided. The aim is to keep techniques complementary, rather than necessary. Similar arguments apply for robust control and common practices in cloud, such as using redundant requests to reduce average latency. The focus of the thesis is on strategies that are generic and complementary to the control problem itself, and useful in a cloud context.

1.3 Thesis outline and contributions

Part I

Part I serves as an introduction to the cloud control challenge. The first chapter defines the cloud, develops the challenge and relates it to networked control. This is followed by an introduction to Model Predictive Control (MPC), which is used to implement control in the cloud. Finally, the plant that is used throughout the thesis is introduced and motivated.

This part contributes with perspective, positioning the cloud control challenge as an intrinsic part of intelligent systems in the cloud. It proposes controller offloading and elastic control as topics for control research.

Part II

Part II examines cloud performance through the application of MPC. Chapter 7 serves to validate concepts and assumptions. It shows the availability of reliable, wireless, low latency communication, and that a distributed cloud is a functional execution platform. The chapter also asserts the reference plant as a reasonable tool for study, and uses a Platform-as-a-Service to demonstrate relocating an executing controller, while it remains in control of the plant. While individual parts (connectivity, software platform, IoT devices, cloud) in Chapter 7 are built using COTS components, the platform as a whole is custom-made. Chapter 8 follows with a closer look at offloading, and the contemporary cloud, using only standard, accessible, tools. These chapters provide insight on what to expect from the software and execution platforms.

This part contributes a research testbed, with next-generation wireless connectivity and a distributed cloud architecture. It provides results on cloud performance, through benchmarks and prototypes. Insights are gained into the use of cloud native software in the distributed cloud, as opposed to using lower level protocols and local networks. An important contribution is the outlook on achievable performance, using novel software design, and the differences between platforms. The section also proposes software migration for control systems and develops this idea into the offloading architecture.

Part III

Part III develops strategies that use the special characteristics of clouds to an advantage, while providing graceful degradation to handle failures. Chapter 10 provides the implementation of a variable horizon MPC, showing the use of utility computing, and parallelism, to relieve the controller from static configurations. Based on an implementation with nominal closed loop stability and recursive feasibility, the controller is examined when switching horizons and entering a local control mode, if conditions require. Chapter 11 introduces frequency switching, through a hierarchy in which a controller is automatically augmented by a higher performing alternative. The final chapter, Chapter 12, extends the control problem of the MPC with explicitly evaluated recovery paths. The recovery paths are created from a recovery mode, that provides graceful degradation in case of connectivity issues.

This part contributes three control designs that utilize the processing capabilities of the cloud. The first design focuses on the illusion of infinite compute resources. It is shown how this can be used to implement the variable horizon MPC. Results of simulations are provided, after which a further contribution is to contrast the results with the use of edge clouds. The second design contributes a hierarchical control structure, using three controllers. The first two implement the same MPC at different sampling rates,

to provide rate switching. The third implements unconstrained control, used to handle situations where neither of the two MPCs provide a control signal. Various ways of combining the modes are examined and the structure is experimentally shown to be efficient and reliable. The final contribution, is a design that wraps a client controller into a framework that can provide reliability guarantees. To keep concepts generic, and, as far as possible, use independent components, the strategy wraps the extended controller in a feed-forward framework. The thesis provides theory, implementation, and simulations that study the effects of such a design.

Part IV

Part IV uses frequency control and a *go-back-home* mode, to create a resilient, best-effort control system. This part studies the consequences of reduced real-time requirements, cluster sharing, and the interaction between resilient controllers and the cloud. Chapter 13 introduces the resilient design, and examines it through simulations, using data from Part II. Details on the used frequency controller are provided in Chapter 14. Chapter 14 then continues to further evaluate the design, on real cloud services. Mitigating properties and cloud scaling is shown, by admitting several clients into a shared cluster.

This part provides a resilient control design that contributes with several results and perspectives for cloud control systems. One, is the removal of execution time deadlines. Another, is delay mitigation in the offloading structure, implemented using predictions and control frequency adaptation. A third, is the demonstrated usefulness of combining the resilient control system and the elastic cloud. The part also contributes simulations, with parameters based on real-world data. A final contribution in this part are the details of the frequency adaptive controller.

Part V

Part V summarizes the thesis into a conclusion and provides suggestions for future work.

1.4 Publications

The thesis is based on the following publications.

Part I

Abdelzaher, T., Y. Hao, K. Jayarajah, A. Misra, P. Skarin, S. Yao, D. Weerakoon, and K.-E. Årzén (2020). “Five challenges in cloud-enabled intelligence and control”. *ACM Transactions on Internet Technology (TOIT)* **20**:1. ISSN: 1533-5399.

The paper discusses emerging cloud services for connected embedded devices. It outlines five resulting new research directions towards enabling and optimizing intelligent, cloud-assisted sensing and control in the age of the Internet of Things. The problems are centered around empowering individually resource-limited devices to exhibit intelligent behavior, both in sensing and control, thanks to a judicious utilization of cloud resources together with recent advances in machine intelligence.

Tarek Abdelzaher suggested the five challenges in cloud-enabled intelligence and control. Karl-Erik Årzén and Per Skarin wrote about offloading and closed loop control.

Skarin, P., J. Eker, M. Kihl, and K.-E. Årzén (2019). “Cloud-assisted model predictive control”. In: *IEEE International Conference on Edge Computing (EDGE)*. Milano, Italy, pp. 110–112.

The paper studies seamless control assistance and design of flexible controllers using the edge cloud. It introduces computational offloading with graceful degradation for executing model predictive control using the cloud and illustrates how a cyber-physical system can be improved while keeping the computational cost down.

The paper is an extract of a longer technical report by Per Skarin, who constructed the study. The contribution was compiled by Per Skarin, Karl-Erik Årzén, Maria Kihl and Johan Eker. The four authors were jointly involved in the thought processes initiating the offloading strategies.

Part II

Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2018). “Towards mission-critical control at the edge and over 5G”. In: *IEEE International Conference on Edge Computing (EDGE)*. San Francisco, USA, pp. 50–57. Best Paper Award.

The paper presents a research testbed built to study mission-critical control over the distributed edge cloud. The developed cloud platform provides the means to continuously operate a mission-critical application while seamlessly relocating computations across geographically dispersed compute nodes. The use of 5G wireless radio allows for mobility, and reliably provide compute resources with low latency, at the edge. The primary contribution of this paper is a state-of-the art, fully operational testbed showing the potential for merged IoT, 5G, and cloud. The paper also provides an evaluation of the system, while operating a mission-critical application, and an outlook on a novel research direction.

The 5G testbed project was proposed by Maria Kihl and William Tärneberg. Karl-Erik Årzén proposed the process and MPC controller. Per Skarin and William Tärneberg implemented the testbed, with William focusing on network integration, and Per on the controller and observability. Problem definition, architecture, and evaluation efforts were shared equally. The illustrations of the testbed were done by William Tärneberg. Maria Kihl and Karl-Erik Årzén continuously assisted the implementation and writing with ideas and directions.

Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2020). “Control-over-the-cloud: a performance study for cloud-native, critical control systems”. In: *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. Leicester, UK, pp. 57–66.

The paper evaluates a set of cloud platforms and infrastructures with the intention of hosting feedback control systems. It shows potential and limitations, by evaluation several protocols and levels of the cloud software stack. It proceeds to evaluate an offloading control strategy, and shows how the sensitive nature of control can cause a seemingly adequate cloud platform to pose a high risk, while a seemingly inadequate platform can have a positive affect on the performance of the controller.

The performance study was suggested, implemented, performed and evaluated by Per Skarin and William Tärneberg. Per, with a broad scope on the software, measurements, and interfacing and William with a stronger focus on protocols and Kubernetes. Maria Kihl and Karl-Erik Årzén assisted in setting the scope for the study. The paper was written by Per Skarin and William Tärneberg with support from Maria Kihl and Karl-Erik Årzén.

Part III

Skarin, P., J. Eker, and K.-E. Årzén (2020). “Cloud-based model predictive control with variable horizon”. In: *21st International Federation of Automatic Control (IFAC) World Congress*. Berlin, Germany.

The paper presents a novel method using the cloud to implement a variable horizon model predictive controller. This is based on the idea that robust, best effort strategies allow industrial grade use of the powerful, efficient, and quickly improving cloud ecosystems. In case of sudden long delays and downtime, a graceful degradation is used. The variable horizon strategy finds use in, for example, non-linear control problems, and the proposed method can be generalized to implement robust and scalable controllers that benefit from cloud technology.

The method was suggested by Per Skarin and Karl-Erik Årzén. Measurements and simulations were performed by Per Skarin. The writing was done by Per Skarin with support from Karl-Erik Årzén and Johan Eker.

Skarin, P., J. Eker, and K.-E. Årzén (2020). “A cloud-enabled rate-switching MPC architecture”. In: *59th IEEE Conference on Decision and Control (CDC)*. Jeju, Korea (South).

The paper presents an architecture for cloud-based MPC, consisting of a high rate MPC in the cloud and a low rate MPC on the local device. The system uses the cloud MPC as the nominal controller but switches to local MPC in case of an unresponsive network. The two MPCs are designed to be as similar to each other as possible, except for the sampling rate. Different alternatives for when to execute the local MPC and how to perform the switching are presented. The approach is evaluated by simulation.

The method was suggested by Karl-Erik Årzén, and developed by Karl-Erik Årzén and Per Skarin. The implementation and evaluation were done by Per Skarin. The writing was done by Per Skarin with support from Karl-Erik Årzén and Johan Eker.

Skarin, P. and K.-E. Årzén (2021). “Explicit MPC recovery for cloud control systems”. In: *60th IEEE Conference on Decision and Control (CDC)*. Austin, TX, USA.

The paper presents a strategy for failure-resilient cloud control using MPC extended with explicit recovery. Based on an arbitrary and unmodified client controller, the remotely operated MPC can safely manipulate the network controlled plant through temporary adjustments of an error signal generator. The paper shows how to implement the reliable cloud controller, relate it to robust MPC and discuss stability. Simulations illustrate the obtained performance and resilience to failure.

The method was suggested by Per Skarin and developed jointly with Karl-Erik Årzén. Theory, implementation and evaluation was done by Per Skarin. The writing was done by Per Skarin with support from Karl-Erik Årzén.

Part IV

Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2021). “Factory automation meets the cloud: Realizing resilient cloud-based optimal control for Industry 4.0”. *Submitted to IEEE Transactions on Industrial Informatics*.

The paper constructs a frequency-adaptive offloaded controller, which can successfully and continuously take advantage of any of a set of cloud deployments. The solution relies on continuously adapting the controller to the prevailing conditions in the cloud. As results show, this allows a control system to survive interruptions, load disturbances caused by other users, and time-varying resource availability. Experiments show a system that can seamlessly switch between clouds and that multiple controllers using shared resources consequentially self-adapt so that no controller fails its objective.

The method was suggested by Per Skarin and developed jointly with William Tärneberg. The cloud infrastructure was developed by William Tärneberg and the control software by Per Skarin. The theory and methods were primarily authored by Per Skarin and the experiments were done mainly by William Tärneberg.

Publications that are not in the thesis

Årzén, K.-E., P. Skarin, W. Tärneberg, and M. Kihl (2018). “Control over the edge cloud - an MPC example”. In: *1st International Workshop on Trustworthy and Real-time Edge Computing for Cyber-Physical Systems*. Nashville, USA.

Tärneberg, W., P. Skarin, C. Gehrman, and M. Kihl (2021). “Prototyping intrusion detection in an industrial cloud-native digital twin”. In: *22nd IEEE International Conference on Industrial Technology (ICIT)*. Vol. 1. IEEE. Valencia, Spain, pp. 749–755.

1.5 Notation

The symbol $t \in \mathbb{R}$ is reserved for denoting continuous time and we let $k \in \mathbb{Z}^*$ denote discrete time indexes. With a recurrent sampling time given by T_s , $x(k)$ is the state at time $t = kT_s$. The letters i and j are used as sequencing variables. When necessary, time and sequence indexing uses subscript such that $x_k = x(k)$. In all subscript notations, t , k , i , and j are indexes, while other symbols are part of the variable name. For instance, the variable x_e may be time indexed as $x_{e,k}$ which should be interpreted as $x_e(k)$.

Sequences can be denoted in bold so that $\mathbf{v} = [v(0), v(1), \dots]$. When unambiguous, this notation is not used. The dimension of variables is given in their specification, such that $x \in \mathbb{R}^n$ is a real vector of length n and $A \in \mathbb{R}^{n \times m}$ is a real matrix of n rows and m columns. The plant state is denoted x and the control signal u . The number of states are n and the number of inputs m . The variable \hat{x} is used to denote the observed (or estimated) state of the plant. \tilde{x} is the error in the observation. $\hat{x}(t)$ is an estimate of the state using

all information available up until time t . A model prediction of the plant state is denoted x_m .

$v(k)$ denotes the value of v at time step k and $v(k+i|k)$ is the predicted value of v at i time steps after time k . This can also be written as $v_{k+i|k}$, or, if \mathbf{v} is a sequence of predictions, as $v_k(i)$, where k is the time of the prediction, and i indexes a position in this sequence. A time sequence is denoted using the vector notation and time index as $\mathbf{v} = [v(k|k), v(k+1|k), \dots, v(k+i|k)]$. Sets are denoted using curly braces as in $\mathbf{v} = \{v(0|\Omega_0), v(1|\Omega_1), \dots, v(i|\Omega_i)\}$ where the index i is the position in the set and Ω_i represent a configuration. The configuration Ω_i provide parameters to create the element $v(i)$. The set is not ordered in relation to the configurations Ω_i unless specifically stated.

The notation $M \geq 0$, $M > 0$ implies that the matrix M is positive semi-definite and positive definite. I_n denotes the $n \times n$ identity matrix.

A sequence can also be represented using superscript such that $\kappa^{0,N}$ is a sequence of $N+1$ functions, indexed from zero. κ^j denotes the j^{th} function in the sequence, and $\kappa^j(x_k)$ evaluates this function with parameter x_k . A subscript is part of the variable name in this notation so that a sequence of control inputs u_r is denoted $u_r^{0,N}$.

Glossary

This glossary includes a small subset of domain specific terminology that is used within the thesis. Its purpose is primarily to avoid confusion in some specific cases.

Cloud Control System (CCS) The term refers to a system where part of the control loop is implemented in the cloud. In the thesis this generally refers to a cyber-physical system where the plant is a physical device.

Control over the Cloud (CotC) Because the term Cloud Control System may be confused with systems that control the cloud itself, the more specific Control over the Cloud is introduced in the thesis to distinguish a separation of the cloud and the system under control.

Cloud Control Refers to the combination of feedback control systems and clouds in general. When unambiguous, this term may be used in place of the more specific phrases control of the cloud and control over the cloud.

Cloud Region A geographical area where cloud services are hosted, such as **us-west** on the United States west coast (California) and **eu-north** in northern Europe (Stockholm, Sweden). A region can also provide availability zones, representing isolated data center locations within the region.

Control of the Cloud Use of control techniques to implement dynamic resource management in the cloud infrastructure (see for instance [Kihl et al., 2008; Tärneberg et al., 2017b; Nylander et al., 2020]).

Execution Time In the presence of a real-time system, the execution time is usually referring to the time a task uses CPU and memory. Wait states, are not part of this time. To ensure that the execution time is not confused with the wall clock time, the term processing time is used in the thesis .

Horizontal Scaling Adding or removing resources used by an application or made available in a cluster, by adding or removing virtual machines (or containers).

Hyperscale Data Center A hyperscale data center has an excess of resources and the ability to scale quickly even in response to a very large in-

crease in demand. To an ordinary user, there are seemingly limitless resources available.

Microservice Microservices are part of the software engineering strategy for cloud applications. A microservice is a small and independent service that communicate over well defined APIs. Components in a microservice architecture are owned by a team of developers who implement, maintain and deployed them, without affecting the functioning of other services.

Processing Time To avoid confusion with execution time and other response times, processing time specifically refers to the response time of a task executing on a computer. That is, the wall clock time span from the instance a task is scheduled to start executing until it has completed. The response time does not include transferring input and output to and from the task.

Response Time Can be used for many different purposes. The response time of the controller, the response time of a request sent over the network, the response time of an application executing in a cloud cluster etc. The response time is always measured in real-time, i.e., as the wall clock time of the system.

Vertical Scaling Changing the configuration of a virtual machine by adding or removing (scale up/down) compute resources, such as the available memory, CPU time, or number of CPU cores.

Wall Clock The real time in the system. Often, the wall clock time refers to the theoretical global time but can, if specified, also be a measurement from the real time clock of a single computer. The wall clock is in contrast to for instance the CPU clock which measures the time spent using the CPU. An application can briefly access the CPU, using very little CPU time, but its run time from start to finish can be extensive in terms of the wall clock due to interruptions or waiting for input-output operations.

Worker Node A machine in the cloud that hosts components of the application workload.

Acronyms

2-DoF	2-Degrees-of-Freedom	169
5G	Fifth Generation Wireless Specifications	58
ADC	Analog to Digital Converter	90
API	Application Program Interface	39
ARPANET	Advanced Research Projects Agency Network	12
AWS	Amazon Web Services	86
CCS	Cloud Control System	46
CFS	Completely Fair Scheduler	90
CGI	Common Gateway Interface	79
CLRE	Closed Loop Response Error	204
CNCF	Cloud Native Computing Foundation	40
CotC	Control over the Cloud	55
COTS	Commercial Off-the-Shelf	77
CPS	Cyber-physical System	56
CPU	Central Processing Unit	38
DAC	Digital to Analog Converter	90
DC	Data Center	78
DNN	Deep Neural Network	47
DNR	Distributed-NodeRED	79
EMA	Exponential Moving Average	209
ERDC	Ericsson Research Data Center	86
FaaS	Function-as-a-Service	37
FPGA	Field-Programmable Gate Array	78

GSM	Global System for Mobile Communications	12
HTTP	Hypertext Transfer Protocol	101
IaaS	Infrastructure-as-a-Service	36
IBM	International Business Machines	16
ICMP	Internet Control Message Protocol	101
ICT	Information and Communication Technology	79
IIoT	Industrial Internet-of-Things	190
IP	Infrastructure Providers	106
ISP	Internet Service Provider	93
IT	Information Technology	39
IoT	Internet-of-Things	35
K8S	Kubernetes	101
LAN	Local Area Network	79
LQ	Linear Quadratic	68
LQR	Linear Quadratic Regulator	56
LTE	Long Term Evolution	84
LuMaMi	Lund Massive MIMO	85
MAC	Medium Access Control	85
MIMO	Multiple-Input-Multiple-Output	85
MLE	Maximum Likelihood Estimation	193
mMTC	Massive Machine Type Communication	84
MPC	Model Predictive Control	48
MQTT	Message Queue Telemetry Transport	39
MU-MIMO	Multi-user MIMO	85
NCS	Networked Control System	48
NFV	Network Function Virtualization	78
NIST	National Institute of Standards and Technology	36
NMT	Nordic Mobile Telephony	12
OSI	Open Systems Interconnection	101
OS	Operating System	37
PaaS	Platform-as-a-Service	36
PI	Proportional and Integral	202

PID	Proportional, Integral and Derivative	46
PLC	Programmable Logic Controller	54
QP	Quadratic Program	67
QoS	Quality of Service	38
RAE	Relative Accumulated Error	103
RAT	Radio Access Technology	85
RAV	Relative Accumulated Violations	102
RBS	Radio Base Station	84
R-CCS	Resilient Cloud Control System	207
REST	Representational State Transfer	99
RMCV	Relative Maximum Constraint Violation	103
ROS	Robot Operating System	18
RTT	Round Trip Time	92
SaaS	Software-as-a-Service	36
SDHI	Software Defined Hardware Infrastructure	38
SDN	Software Defined Network	40
SMC	Switching Multitier Control	127
SUNET	The Swedish University Network	100
TCP	Transmission Control Protocol	101
TSN	Time Sensitive Networks	17
UDP	User Datagram Protocol	101
UE	User Equipment	78
URLLC	Ultra-reliable and Low-latency Communication	54
VM	Virtual Machine	37
VPN	Virtual Private Network	87
vSoftPLC	Virtual Software Programmable Logic Controllers	79
WAN	Wide Area Network	82
WAP	Wireless Application Protocol	12
WASP	Wallenberg AI, Autonomous Systems and Software Program	86
WWW	World Wide Web	11

Part I

Cloud Control Challenge

2

Intelligence in the Cloud

If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility... The computer utility could become the basis of a new and important industry.

John McCarthy speaking at MIT in 1961
[Simson L. and Harold, 1999]

The broad availability of cloud services offers an opportunity to extend the capacity of connected devices through remote access. Collections of Internet-of-things (IoT) devices often transfer their status to applications implemented in clouds, for monitoring and analysis, to perform anomaly detection and measure performance. The knowledge and actions that are deduced from the collected information can be fed back over the network to implement feedback control - often referred to as *closing the loop*. This connectivity also opens up a communication path for control client software to use services in the elastic service infrastructure (Section 2.1). Because the underlying cloud platform is scalable, it can be used to launch subsystems that coordinate a group of devices, or deploy heavy computations as needed.

Figure 2.1 provides an example. The cameras in this figure can independently and jointly monitor an area of interest, but they also send data to the cloud for storage and advanced processing. This established connection to the cloud can also extend the capabilities of the cameras. There are several ways in which this can be used. A camera can request support to analyze a stream of pictures, a remote supervisor can reconfigure the group to improve their performance or handle an unusual situation, or the cloud can take direct control of the system, directing the cameras as a single unit to obtain optimal performance. The resource requirements in different situations varies, making the *scale with demand* property of clouds very useful. When the sensing,

Parts of this chapter are based on [Abdelzaher et al., 2020] and [Skarin et al., 2019]

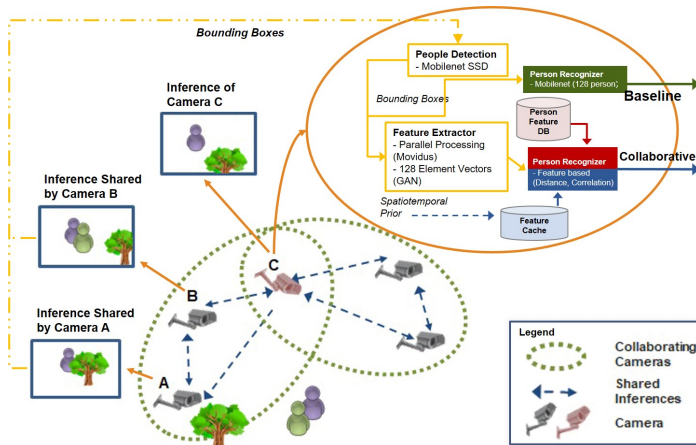


Figure 2.1 Collaborative IoT (Camera) environment & Deep Inferencing Pipelines.

decision making and learning processes of autonomous systems are moved to the cloud, it is referred to as placing *intelligence in the cloud*. The following chapter serves as an introduction to what the cloud is and how closed loop control is implemented in the thesis.

2.1 Cloud Overview

The most commonly referred to definition of cloud computing is the one from the United States National Institute of Standards and Technology (NIST):

Cloud computing is a model for enabling *ubiquitous, convenient, on-demand* network access to a *shared* pool of *configurable* computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly *provisioned and released* with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. [Mell and Grance, 2011]

The definition mentions five essential characteristics, these are: *on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service*. The thesis' relation to the essential characteristics is presented in the next section. The definition also mentions three service models, they are: *Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS)*. The definition of these services will

be returned to shortly, and the *Function-as-a-Service (FaaS)* will also be introduced.

Elastic utility computing

Two characteristics that are of special interest to control theory and control systems engineering is resource pooling and rapid elasticity. They are defined as follows by NIST:

Resource pooling The provider’s computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.

Rapid elasticity Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

We will henceforth assume that resources are necessarily pooled and refer to the flexible and rapid provisioning and reconfiguration of cloud resources as *elasticity* and *utility computing*. The definition of rapid elasticity above mentions outward or inward scaling. Scaling is often separated into scaling in and out through horizontal scaling, or scaling up and down through vertical scaling. The former adds more machines to a cluster, while the latter re-configures machines with new capabilities. To implement these features, clouds use two virtualization technologies: Virtual Machines (VMs) and containers. The two technologies are illustrated in Figure 2.2 and introduced below.

Virtual Machines

Full virtualization, provided through VMs, use software to create isolated environments that allow complete operating systems (OSs) to co-exists on the same hardware. Network interfaces, graphics processing units, disks, and so on, are replaced by software devices to create virtual computers. As seen in Figure 2.2, on top of the hardware there is a host, which is the ordinary OS. Next to the hosts there is a *hypervisor*. The VMs execute through the hypervisor. A hypervisor emulates necessary instructions to ensure that the guest OS kernel inside the VM works properly. This is necessary, because the guests OS kernel assumes it has exclusive privileged access to the physical

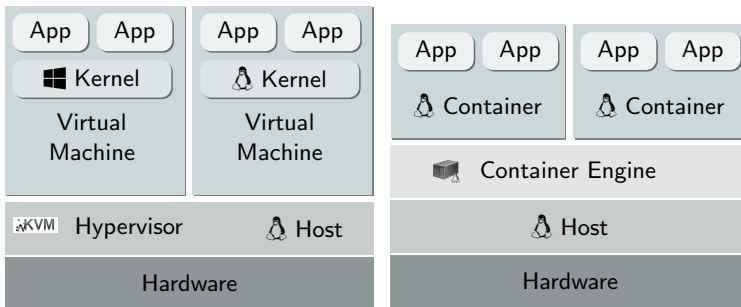


Figure 2.2 Virtual machine and container stacks.

machine. In this setup with a hypervisor, several users can execute completely different OSs in parallel on the same hardware.

Modern computers, even ordinary personal computers, implement support to accelerate the execution of virtual machines. This makes memory and CPU intensive tasks virtually indistinguishable between the host and the VM. A VM can be configured to access a sub-set of the physical machine's resources, CPU, memory, etc. If necessary, the VM can later be reconfigured with a new set of resources. A VM can also be moved (migrated) to a different physical machine while the guest OS remains on-line. Migration of VMs is a slow, incremental process that can take many minutes to complete and can cause quality of service (QoS) degradation and even downtime [Sapuntzakis et al., 2002; Ahmad et al., 2015]. While the maximum number of Central Processing Unit (CPU)s and the amount of memory available to a single machine is limited by what is available on a single mother board, the storage for virtual machine disks can be served from a distributed storage pool (such as Ceph [Weil et al., 2006]). This truly makes storage seem virtually infinite. In future systems, Software Defined Hardware Infrastructure (SDHI) [Roozbeh et al., 2018] aims to implement a single 'infinite' pool also for memory and CPU.

Containers

Containers implement operating system-level virtualization, often referred to as light-weight virtualization. The concept was pioneered by Sun Microsystems as Solaris Containers in 2005 but became popular many years later on in the Linux OS. As seen in Figure 2.2, containers share the OS kernel of the host and a hypervisor is therefore not needed. Containers package their own files, libraries and utility software, i.e. the user space content of the OS, but system calls (i.e. kernel code) execute directly in the host's privileged memory space.

The host kernel implements features that allow for a container engine

to setup separate networking, file system structures, memory and CPU restrictions, etc., for the individual containers. Containers do not allow for completely different OSs to co-exist but they support, for instance, different distributions of Linux and a variety of user space components that are compatible with the host kernel. Containers provide a very practical platform as they simplify version management and software deployment, and, while not as isolating as a VM, they provide a large degree of isolation. Containers are used to package microservices, and container engines are often deployed inside virtual machines.

Services

The cloud provides networked services to simplify software development and infrastructure maintenance. In the classic perspective, clouds provide three categories of services: IaaS, PaaS, SaaS. A fourth category, FaaS, is introduced with the contemporary cloud in Section 2.2.

At the lowest level, there is IaaS. This service provides access to provision and configure VMs, storage and networks through web browsers and Application Program Interfaces (APIs). The user installs OSs, configures servers, routers and so on, managing a virtual Information Technology (IT) infrastructure inside the data center. PaaS provides a hosted application development and execution platform in the cloud. The PaaS implements convenient ways of deploying and monitoring software, supporting continuous integration and DevOps practices. The user of a PaaS is limited to the capabilities of the platform, its method of scaling, logging etc, but does not have to manage the IT infrastructure. SaaS is at the highest level and provides applications such as email and word processing as a service accessed in a web browser.

IaaS has been used extensively to implement the higher level services that are used in the thesis, and a PaaS is also present. Overviews and some details around these architectures will be revealed, but it is out of scope for the thesis to go deep into the implementations.

Communication Protocols

TCP/IP networks and HTTP interfaces are important to cloud computing. The convenience of using these two Internet technologies makes them ubiquitous in the community. Internet services are provided over RESTful interfaces implemented using HTTP, and the communication between containers and VMs is implemented as TCP/IP networks. Message passing systems such as Message Queue Telemetry Transport (MQTT) [Naik, 2017] are also common, especially in relation to IoT. This provides a convenient level of abstraction that makes IoT systems and clouds very accessible. The thesis uses high level interfaces and communication over public networks. This can be an inhos-

pitabile environment for sensitive systems but requirements such as interoperability (Section 1.1) and convenience (Section 2.1) make it reasonable that this is our entry point into the cloud ecosystem.

Software Defined Networks

Software Defined Networks (SDNs) are not used per se in the thesis but are worth a mention nonetheless. As made evident by the previously mentioned SDHI (software defined hardware infrastructure), flexibility through software is continuing to evolve in the cloud ecosystem. Not only does virtualization provide software defined network interfaces for virtual machines and containers, the networks connecting the machines in the data center are also programmable. A router is no longer only upgraded with official firmware, it can also be supplied with new routing software from a local administrator. This says something about the involved complexity and flexibility of the considered technology. Even an application that uses a low level transmission protocol, such as UDP, is subjected to SDN, overlay networks (for security and to create virtual local networks), and network interfaces (i.e. network cards) implemented in software. This can also be layered, i.e. a VPN implemented inside a VM, and may or may not implement bypasses to improve efficiency. In general, we have to expect that there are layers of configurable elements for just about everything in a cloud.

2.2 Contemporary Cloud

The cloud ecosystem has grown into a large and complex landscape. There are several public cloud providers with their own interfaces and services, and organizations are also using private and hybrid (public and private) solutions. The Cloud Native Computing Foundation (CNCF) collects a cloud native landscape [CNCF, 2021b] with roughly 900 tools for application definition and development, orchestration and management, runtimes, platforms, provisioning, observability and analysis. The charter of CNCF, started in 2015, presents a mission to make cloud native computing ubiquitous. CNCF, a part of the Linux Foundation, now has more than 650 members, including the most prominent cloud providers. The foundation fosters the use of open source and vendor-neutral components, so its landscape does not include all cloud solutions. Nonetheless, the foundation's definition of cloud native provides a useful amendment to the NIST definition (see Section 2.1).

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes,

microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil. [CNCF, 2021a]

Today, Kubernetes has become the dominant cloud native platform to match the vision of CNCF. Kubernetes is a cluster management platform that promotes the creation of micro services and applications built using containers. A fourth service type, FaaS, has also emerged, in addition to the three classic categories of the cloud stack (IaaS, PaaS, SaaS). As the name suggests, in FaaS the user provides code and/or binaries for a single function and stores it in the cloud. FaaS is event driven, with the function executed in a stateless environment setup by the cloud provider, such as a container providing Python support. A difference between PaaS and FaaS is that the former generally includes an active server process that receives external requests. The PaaS can be stateful and is scaled by booting up more server processes. These servers are visible to the developer who is also charged for their existence.

In FaaS, the function is idle and not charged until an event arrives, requesting that the function processes some input data. The function is launched, executes to completion and provides a response, then goes out of existence, as far as the developer is concerned. The function can be triggered at any time and can be launched as any number of parallel jobs. The developer can configure parameters that effect the execution of functions (such as maximum execution time and maximum number of simultaneous instances) but the cloud provider handles the service scaling. The user of the function service is charged for the compute time consumed by the function. FaaS is a way to achieve cost efficient scalability but comes at the cost of latency. FaaS is sometimes referred to as serverless because the developer does not have to worry about the persistent execution or scaling of machines in the cloud. FaaS is a practical way to implement micro-services and IoT applications. The thesis will deploy Kubeless, a FaaS service, on top of Kubernetes and experiment with the Amazon Lambda function service in Chapter 8. A compatible service is also implement using IaaS. The latter has a traditional architecture and is not as resilient, automated and observable as the cloud native solutions.

Figure 2.3 illustrates this difference and provides some insight into the architecture of a cloud native deployment. The figure shows one project in an OpenStack¹ cloud, with three separate networks of connected machines.

¹OpenStack is the leading open source cloud computing platform

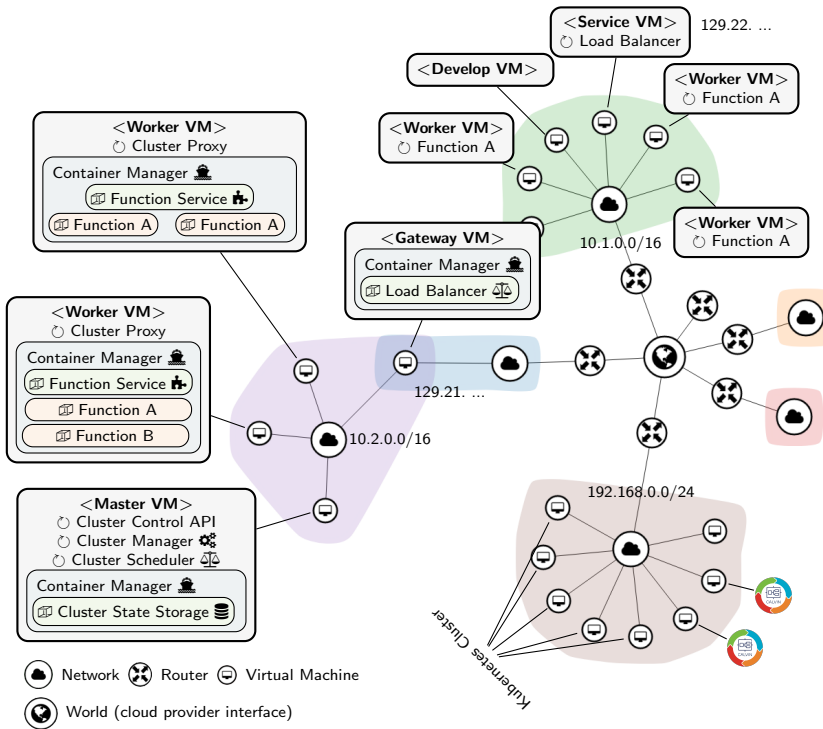


Figure 2.3 Clusters of a project in OpenStack.

The outer boxes in the figure, represent virtual machines. Inside, there are boxes to represent container managers, and inside those, boxes that represent containers. The network topology and virtual machines are configured through IaaS, and containers communicate using network interfaces.

The green cluster, at the top, represents a classic web application setup. There are several workers that can service a user request, in this case only providing Function A. Worker selection is performed by the load balancer, which is the entry point to the cluster. A development machine is present in the cluster. It can be used for various management and prototyping. Access is granted from external sources by the cloud routing a public internet address to the Service VM.

The purple cluster, on the left, shows a cloud native deployment akin to an environment such as Kubernetes. The Master VM hosts a Cluster Manager service that ensures the consistency of the cluster. If the cluster is reconfigured, or some part of the cluster is malfunctioning, the manager works to bring the cluster back to the operator requested state by shutting

down, starting and restarting services. The Cluster Scheduler (also in the Master VM) overlooks the health of the cluster in terms of resources and can provision worker deployments to improve capacity.

Applications are launched as containers handled by Container Managers executing inside the virtual machines. In the figure, a Function Service is deployed as containers in the cluster. It is represented by green boxes in the Worker VMs. The function service in turn manages additional containers that provide the user functions. The Cluster Control API (Master VM) allows developers to easily deploy other services next to the function service, and extract information about the cluster nodes and applications. An external load balancer is placed in a Gateway VM, separating the cluster from the global network.

Additional VMs can easily be requested from the cloud to deploy additional workers as necessary. Several masters can also be active to provide additional resilience and performance. The container manager provides container services for both the cluster and the user applications. In the figure, many cluster functions execute directly in the virtual machine OS but most features can also be placed as containers managed by the Container Manager. This way, the deployment and upgrade of the core system can also utilize the convenient features of the cloud native environment. Each VM can be different and do not need to host equally configured OSs. To upgrade a worker, a new virtual machine can be commissioned, installed and configured, its function verified, and the obsolete virtual machine (the obsolete configuration) removed when done. The same flow applies to containers, and in extension functions, but at a different time scale.

At the bottom there is also a third, pink, network. This network serves a cloud native cluster similar to the purple deployment, but those details are not shown. The network also serves two machines represented by icons displaying a circle of red, green, blue and orange arrows. These machines are part of a Calvin network. They execute a runtime that provides a PaaS that will be detailed in Chapter 7. These virtual machines are part of a mesh network, creating a distributed execution platform spanning the cloud in Figure 2.3, a base station edge break-out², an IoT device, and a second data center. Everything in Figure 2.3 is software defined and exist as virtual resources in the data center. The topology can be reconfigured with resources removed or added from a command line client or a web interface, with changes applied promptly and billing changed to reflect the new resource usage.

To round of this overview of the cloud we turn to Figure 2.4. While the previous iteration of cloud technology evolved the native technology in Figure 2.3, this figure represents a small snapshot of the critical cloud ecosystem that is currently evolving. At the center of the picture we see a private data

²Additional computers used to serve a small edge cloud connected to the base station

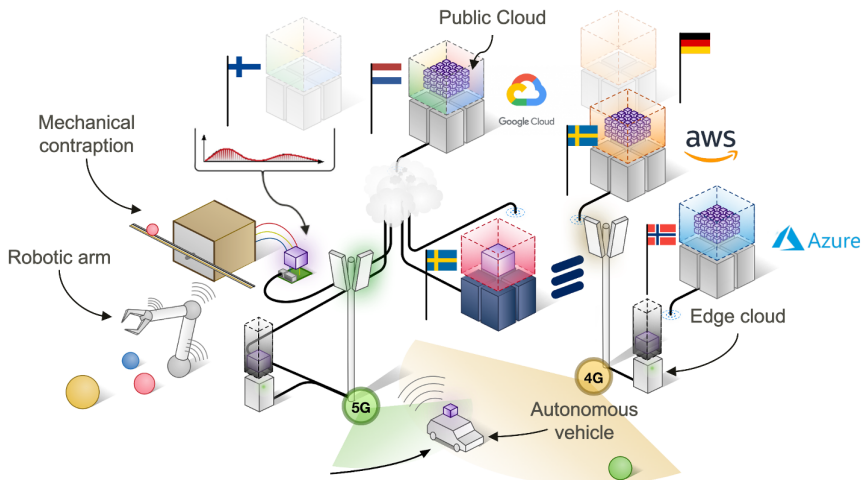


Figure 2.4 The cloud is a ubiquitous part of many systems and is set to become more diverse.

center, represented by the Ericsson logo. Around it there are data centers representing three public cloud providers. There are also smaller edge clouds connected to 5G and 4G base stations. At various locations in this infrastructure, the autonomous vehicle and the robotic arm continuously filter their data through machine learning algorithms, coordinate with other systems and execute real time control to achieve their objectives. We place *intelligence and control* in the cloud to make individual devices exhibit intelligent behavior. The distributed cloud provides services that allow a wise use of the cloud for a large range of non-critical and critical applications. The mechanical contraption in the figure, which will be introduced and used later, is also acting in this environment. While the physical properties, objectives and contexts of various machines are different, the way in which they interact with the ecosystem in Figure 2.4 is the same. Therefore, the contraption is a drop in replacement of the other systems and represents rapid prototyping, experimentation, education, and other positive benefits of the cloud platform.

2.3 Research Challenges

In [Abdelzaher et al., 2020] five challenges were identified for placing intelligence and control in the cloud.

- 1. Learning-as-a-Service** Learning is an important part of autonomous systems, both offline and on-line. How can one implement

learning as a general-purpose service?

2. **Sensing quality assurance** The complexity of sensing and control applications, and the presence of learning components, lead to a very important challenge: how to make guarantees on quality of results? Furthermore, cloud-assisted execution incurs cost, so it is important to understand the trade-offs between incurred resource demand and result quality.
3. **Offloading optimization and control** In intelligent sensing, the cloud can be used to offload or cache machine learning tasks and data, such as computing parameters of a *neural network model*. Much alike, certain closed loop control functions can be offloaded to more capable machines in the cloud.
4. **Closed loop guarantees** A key challenge in placing parts of a control loop in the cloud is to provide some guarantees on closed loop performance, even in the presence of unpredictable latency or loss of connectivity.
5. **Collaborative execution** In many environments, IoT devices are not deployed individually, but rather as a collection of nodes, possibly heterogeneous, that together support an application. In the distributed cloud environment, how can one support such collaborative inferencing?

This thesis focuses on the challenge of offloading optimization and control, and naturally has to consider the tightly coupled closed loop guarantees. A further complication is that, while it may be possible to benefit from offloading to the cloud and provide closed loop guarantees in isolation, the true power and intelligence happens when solutions are automated and the challenges are combined. Ultimately, this leads to flexible solutions that allow plug-and-play integration and scalable designs. It is inconceivable that all current and future risks of executing out of specification are assessed prior to deployment of such a system. The offloading strategy should therefore be flexible and closed loop guarantees resilient to unanticipated events. This is especially true when there is also volatility and uncertainty in the underlying execution platform.

2.4 Controller offloading

A useful way to implement control over the cloud is through controller offloading, i.e. placing part of the computations necessary to form a control decision in the cloud. A distinguishing feature of offloading is that it is initiated from the client. Offloading can be requested to support an action that

the client is otherwise unable to perform, or to relieve the client from processing, reducing energy consumption or releasing resources for other tasks. The client may continuously request input from the cloud but can also initiate offloading for a certain transaction. It is reasonable, that a client which is capable of initiating offloading is also capable of a basic level of control. This is useful in two ways that are intrinsic to cloud control system (CCS):

- 1) utility computing can be used, requesting support as-needed and when cost allows, and
- 2) availability issues, partition tolerance, and request failure are supported.

One question is what type of control that may benefit from offloading. In industrial process control, the basic control is often provided by Proportional, Integral and Derivative (PID) controllers. These controllers require very few computations (e.g., around 15-20 lines of sequential C code is enough for a good PID controller including, the code for the logical safety network). The same holds for control in home automation, where often even simpler controllers are used (e.g., on-off controllers, proportional controllers, or integral controllers). For these types of controllers, computational offloading is not worthwhile.

However, there are a number of control loop examples for which offloading is realistic. One such example is when a (possibly simple) controller interacts with a more complex sensor. This includes controllers based on vision feedback. Compute-intensive image processing may be needed. For example, in a self-driving car, a deep neural network might be used to extract object information from a video stream in order to recognize obstacles, determine positions and trajectories of other vehicles, and finally control the trajectory of the autonomous car. Two further examples are introduced in the following sections.

Learning-based control

The second example is control based on models derived using machine learning. The aim is to make cloud control systems resilient, and resilient systems ultimately allows for exploration, which leads to learning. Furthermore, machine learning can be transient and extremely resource intensive, and resources is something the cloud can provide.

The use of learning techniques in closed loop control has a long history. One of the first examples is *dual control* proposed in the mid 1960s [Feldbaum, 1965]. The term dual refers to the need for the controller to both online identify, or estimate, the model of the process, and to utilize this model to control the process under uncertainty. This trade-off is the same as what is known in the machine learning community as the trade-off between explo-

ration and exploitation. The optimal solution to the dual control problem can be found using value functions and dynamic programming [Bellman, 1957]. Dual control was relatively popular in the control research community during the 1970s but because the approach suffers from the curse of dimensionality this interest soon decayed due to the lack of computing power at the time. Instead the focus shifted to adaptive control formulations based either on approximations or reformulations of the optimal dual control problem.

The interest for learning-based control has exploded during the last 10 years, mainly as a result of the success that reinforcement learning (RL) has had for various applications [Sutton and Barto, 1998]. A major reason for this success is the availability of large-scale compute facilities based on hardware acceleration and cloud technology, which is available now but not during the 1970s. RL has many similarities with dual control. Both frameworks are based on dynamic programming and for both the trade-off between exploration and exploitation is essential, see [Recht, 2018] for a comparison. The major successes of RL have, however, been found for applications where the state space and the action space are discrete, e.g., different game playing applications such as Atari [Mnih et al., 2013], AlphaGo for playing Go [Silver et al., 2017] and AlphaZero for playing chess [Silver et al., 2018]. There the results have been spectacular, by far outperforming the best human players. However, as soon as the state and/or action spaces are continuous the results are, so far, less convincing. Also for discrete domain applications it is very common that the size of the state space is so large that it cannot be represented explicitly, e.g., the size of the state space of Go is $\approx 10^{170}$. Instead, the value functions are approximated using some function approximation method, e.g., a deep neural network (DNN), and then the difference between applications with discrete and continuous state-spaces becomes smaller. Hence, a generic DNN service that supports offloading of machine learning applications for IoT is applicable also to on-line learning-based control.

Optimization-based control

The third example is optimization-based control. In optimization-based control, the control algorithm consists of the on-line solution to an optimization problem. The optimization problem is formulated so that it minimizes some cost function, subject to the dynamics of the process under control and constraints on, e.g., the control signal, the process output, or the process states. This optimization-problem is solved repeatedly at each sampling instant. Depending on the process model, the cost function, the plant's state, system constraints and the selected solver, the optimization can be more or less time-consuming.

The computational demands depend on both the nature of the underlying control problem and the related safety and performance requirements.

A common case consists of a linear controlled process and an optimization problem with a quadratic cost function and linear constraints. This gives rise to a quadratic programming problem for which very efficient solvers are available. Yet, even with state-of-art solvers, the computation time may vary substantially depending on whether the constraints are active or not. In the case of nonlinear processes, either gain-scheduling between a set of linear optimization problems can be used or a nonlinear optimization problem must be formulated. The latter can be very time consuming and require more compute resources than what is available on local devices.

Optimization-based control is an increasingly popular technique. The most commonly used form of optimization-based control is Model Predictive Control (MPC) [Rawlings and Mayne, 2009b]. MPC is used in the thesis and is introduced in detail in Chapter 4.

Challenges and opportunities

A general problem with offloading controllers is the increased latency from the sensing to the actuation that it incurs. Control performance and stability crucially depend on the experienced latency and jitter. The longer the latency, the worse the performance. It is often possible to partially compensate for the effects of the latency, but it can never be completely undone. As mentioned in Section 3.1, networked control systems (NCSs) make assumptions about the latency in order to allow for analysis and formal guarantees. To successfully implement a practical control over the cloud, we cannot make such assumptions.

A second problem with offloading, in particular when wireless networks are involved, is the risk of completely losing the connectivity between the local device that is connected to the process under control and the node to which the computations have been offloaded. Many control applications are mission-critical and require that the controller promptly reacts to disturbances and commands (e.g., changes in the setpoint or reference signal). It is therefore essential that the control that is offloaded and is executing remotely (e.g., at the edge or in a remote data center) is complemented with a local controller that is able to provide some basic level of performance, in case of connectivity loss. Related to this, there is a question of when to switch between the local and remote controllers, and what information the decision should be based on. An additional complexity is created by dynamic changes in the latency characteristics, caused by migration of the offloaded computations between different nodes in the network, as an effect of load balancing and other cloud artifacts. This means that latency will not only vary due to varying computation times (e.g., in an optimization algorithm) and communication delays (e.g., due to packet collisions), but also due to changes in the placement of the computations and to varying request admission times.

However, using the cloud for offloading also has several advantages. The illusion of infinite compute and storage resources that the cloud and the edge/fog provide opens up a number of interesting possibilities for control applications. The resources can be used for executing more advanced control strategies (e.g., based on online optimization and learning using massive data sets), than what is possible on the local device. The cloud can scale resources with the problem and implement efficient strategies for each computation. This allows the controller to evaluate complex problems that are too computationally demanding to perform locally. Information made available through the communication network (e.g., additional more complex models and information about other similar application types) can be incorporated and used to improve the control, avoiding the overhead and potential concerns of communicating this information to the local device.

2.5 Closed loop guarantees

A key challenge in controller offloading is to provide some guarantees on closed loop performance, even in the presence of unpredictable latency or loss of connectivity. Stability often requires that the control loop remain closed. The general solution is thus to use a local controller that can take over from the remote controller when communication is lost or the latency is too long. A challenge is to understand the conditions under which such a hybrid scheme might offer stability assurances. Beyond stability, one can also consider other guarantees, such as those on worst-case response time, maximum overshoot, or worst-case settling time.

Control theory offers rich literature on techniques used to attain the aforementioned guarantees in the context of controlling both physical [Dorf and Bishop, 2011] and computational [Hellerstein et al., 2004] systems. Most of that literature assumes sufficient connectivity between the control algorithm on one hand, and the sensors and actuators on the other. When the controller is in the cloud, this basic connectivity assumption underlying (most) existing literature may be violated. Lack of predictable connectivity makes it challenging to offload control algorithms away from the controlled system.

3

Control over the Cloud

This chapter provides a short introduction to networked control, real-time, and cloud control systems, as they are related to the contents of the thesis. This is followed by an introduction of Control over the Cloud and the offloading control used to implement experiments in the thesis.

3.1 Networked control

Networked Control Systems (NCSs) are control systems where parts of the control loop are separated by an unreliable communication channel. This communication introduces either or both, non-negligible delays and packet dropout. Two general classes of NCSs can be considered: direct networked control and hierarchical network control. In the former, the control signal is transferred directly over the network to the actuator. In the latter, additional control logic is present on the plant side of the network. A basic representation of these two structures are shown in Figure 3.1. The details and terminology of the structures differ in the literature (see for example [Tipsuwan and Chow, 2003; Zhao et al., 2015]). For instance, the lower part of the hierarchical structure to the right in Figure 3.1 may be referred to as

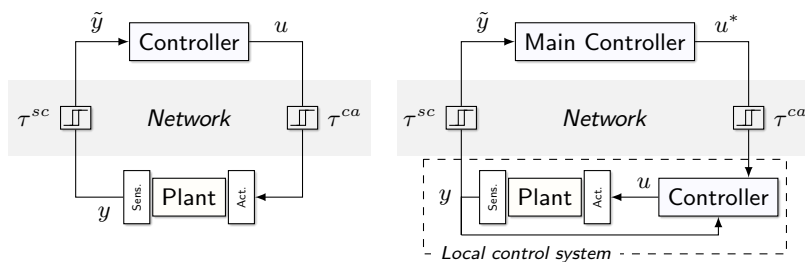


Figure 3.1 Direct and hierarchical network control

the local or remote control system, depending on the perspective. To avoid confusion, this end of the network is referred to as the client, from the client-server model. When the term local is used, it refers to something that is local to the plant, unless otherwise specified. With this perspective, the remote is always something that executes in the cloud.

A large part of the literature on NCSs deal with the direct form to the left in Figure 3.1. One reason is that the local side of the hierarchical structure can be represented by a new model, and the following analysis can be done using the direct form. As mentioned in Section 2.5, most networked control must nonetheless restrict the uncertainty of the network to ensure that control actions repeatedly arrive at the plant, to ensure stability and performance. The delays in Figure 3.1 are represented in state space form as,

$$\begin{aligned}\dot{x}(t) &= Ax(t) + Bu(t - \tau^{ca}) \\ y(t) &= Cx(t) + Du(t - \tau^{ca}) \\ \tilde{y}(t) &= y(t - \tau^{sc})\end{aligned}$$

with τ^{sc} the sensor-to-controller delay, τ^{ca} the controller-to-actuator delay, and where A,B,C,D are the state space matrices for the plant without delays. With constant delay, this representation allows for straight forward analysis and an analytic definition of a controller. The form however, does not allow packet loss. To handle stochastic delay and drop outs, delay distributions, statistical expectations, jump linear systems, and packet drop out processes modeling are introduced (see for instance [Nilsson, 1998; Posthumus-Cloosterman, 2008]). Control problems that incorporate such information achieve better performance than deterministic alternatives, but also depend on more information about the environment.

Generally, the range of network delays must be limited with known distribution, and assumptions such as those in [Tipsuwan and Chow, 2003] apply, i.e.,

- network transmissions are error-free,
- every frame or packet always has the same constant length,
- the computational delay of the controller is constant and is much smaller than the sampling period,
- the network traffic cannot be overloaded, and
- transmission sizes of measurements and control signals are small.

In addition, one controller is supplying the control actions and can make assumptions about what is applied to the plant. These are the kinds of assumptions that we do not make in the cloud control system.

A common representation of delays is shown in Figure 3.2. This picture includes a processing delay, τ_k^c , in addition to the sensor-to-controller, and

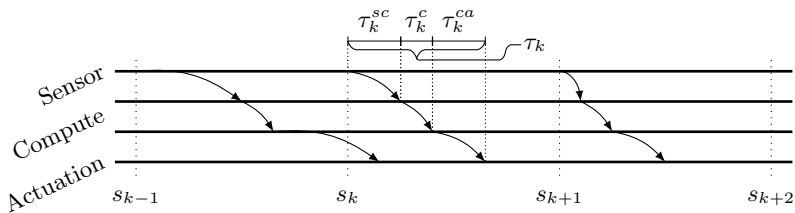


Figure 3.2 The basic delay constituents.

controller-to-actuator delays present in Figure 3.1. The representation in Figure 3.2 also extends to time varying delays, as indicated by the parameter k . The sampling time in this illustration is fixed at $h = s_k - s_{k-1} = s_{k+1} - s_k = s_{k+2} - s_{k+1}$. The control signal generated from sample s_{k-1} arrives after time k , and the sensor-to-actuator delay is $\tau_{k-1} > h$. The control signal from the sample at s_k arrives before $k + 1$ and $\tau_k < h$. From works such as [Bo Lincoln, 2000] it is established that the latter is referred to as short delay, while the former ($\tau_k \geq h$) is an example of long delay. These classifications cover enough detail for most control system analysis. It is also common to work with the combined sensor-to-actuator delay, τ_k and study problems that include only the sensor-to-controller or sensor-to-actuator delay.

Figure 3.3 shows a schematic view of the direct networked control system, where prediction is used to handle network delay, as presented in [Tipsuwan and Chow, 2003] based on the original design in [Luck and Ray, 1994]. The loop introduces a deterministic delay through the predictor and a queue. In this loop, the observer is reconstructing the plant state using the input memory in Z_k . This observation is passed to the predictor, to predict the future state $\hat{x}_{k+\mu}$ using a model of the plant. The predicted state is input to the controller, which produces the control signal for time $k + \mu$. A queue at the other end of the network ensures timely input of the control signal.

In an implementation, assuming packets in sequence and no packet drop-out, the observer and predictor takes the form

$$x_{k+1} = Ax_k + Bu_k; \quad y_k = Cx_k \quad (3.1)$$

$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k + L_k(y_k - C\hat{x}_k) \quad (3.2)$$

$$\hat{x}_{k+\mu} = A\hat{x}_{k+\mu-1} + Bu_{k+\mu-1} \text{ for } \mu \geq 2 \quad (3.3)$$

where Equation (3.1) is the plant model, Equation (3.2) the observer model and Equation (3.3) the predictor. This form handles delays that are multiples of the sampling period. Additional care and further details are needed if packets can arrive out-of-sequence, or if drop-outs may occur.

With the exception of Chapter 7, a single step prediction is consistently used in the thesis. This is part of the scope in Section 1.2, but results should

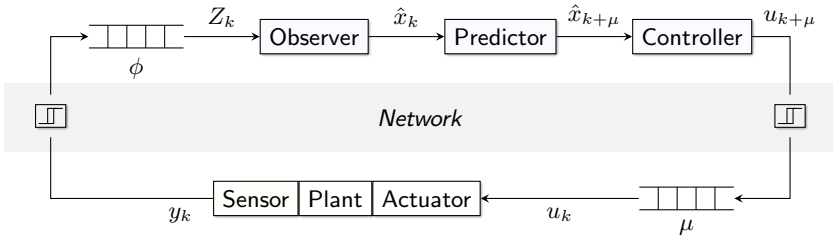


Figure 3.3 Deterministic predictor-based delay compensation.

be easy to transfer to a setup similar to Figure 3.3.

3.2 Real-time

In computing, a system with hard real-time guarantee responds to events within a predictable time. In time-triggered systems this implies well defined sampling and actuation times, often at a fixed rate, i.e., the sampling time. Deviations from the assumed periodicity, or the ideal timing, is referred to as sampling and actuation jitter. The dependence on real-time systems in the control community has since long been established and deterministic execution is often assumed. In [Årzén, 1999] it was argued that a reliance on well performing but non-deterministic off-the-shelf systems, networked control, and hybrid controllers, makes this assumption unrealistic. Back in 1999 the problem related to execution on personal computers, loops closed over local area computer networks, and switching controllers that may change sampling interval in a controlled fashion. In a cloud context, the complexity is far greater, and the problem exacerbated, but the underlying positive aspects of applying off-the-shelf technology and relieving systems from the strict real-time requirements remain the same.

The term *off-the-shelf* is reiterated, and how it appears in the context of the thesis. In previous works such as [Årzén, 1999; Seto et al., 1998] off-the-shelf refers to the use of general purpose networking and computers, operating systems, and advanced middleware. While these system do not exhibit ideal characteristics, they can be deployed and thoroughly tested, after which they remain unchanged, or are updated in a highly controlled fashion. In the cloud domain, we extend off-the-shelf to include cloud services, virtual components, software defined networks, open-source-software, and IoT devices. Many of these are shared components that are usually maintained independently of our specific control loop. Therefore, execution characteristics can change during run-time, due to load etc, and a controlled update, such as

a software security fix, may introduce side effects to performance. Besides, clouds today, and the typical communication medium to access them, do not include real-time support. A mobile edge cloud compute node with access through ultra-reliable and low-latency communication (URLLC) can provide deterministic response times. This is less likely for execution in a data center, accessed over several shared networks. Improved support for real-time systems is likely to be implemented in broadly available clouds and networks, but here, this is not an assumed property.

3.3 Cloud control system

The structure to the right in Figure 3.1 and the structure in Figure 3.3 form a basis for the feedback control systems in this thesis, but there are differences. Control signals from the cloud do not necessarily pass through the local controller, but instead replaces its control actions. The MPC also provides open loop predictions and control signals that can be sent to the client. In this view, the CCSs is an extension of a predictive NCSs. Others have chosen this view, [Mahmoud and Xia, 2020; Vick et al., 2016], and applied the predictive control system as the solution to the problem of stochastic networks.

Elastic control

While it may be useful to implement traditional control systems in the cloud, they do not necessarily have to be considered as cloud control systems. Often, the implementation, assumptions, and analysis used in control over the Internet represents the deployment of a NCSs. This has been extended to include virtual machines, with no modification of the controller implementation and analysis. As a direct extension of control over the Internet, experimental works implement a programmable logic controller (PLC) [Givehchi et al., 2014] in a virtual machine or control drones through a series of cloud services [Pelle et al., 2019]. These works provide suggestions and isolated studies of the system to the left in Figure 3.1, but fall short in two useful aspects: the control loop does not implement fallback and graceful degradation, and the controller does not autonomously benefit from utility computing.

Another class of control systems emerges when we move away from a classic, static view and transfer the elastic property of clouds into the control loop. In this view, the control loop should aim to use the abundant resources of the cloud *as necessary* and *when available*. On-going work to provide low-latency and predictability in networks and clouds is not in conflict with this view. It is an intrinsic feature of the elastic control systems that they make use of iterative infrastructure improvements, and there are several other benefits of designing with elastic properties in mind:

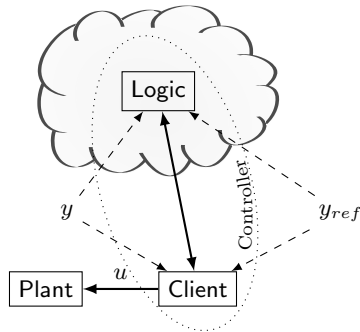


Figure 3.4 Control-over-the-cloud

- 1) an ever so small risk of an inaccessible remote system no longer has to be a major concern,
- 2) external events not caused by the communication network or cloud system provider can be acceptable, for instance a configuration change or application software upgrade,
- 3) graceful degradation opens up for flexibility such as physical mobility of the client, software relocation and on-line upgrades, and
- 4) resilient design opens up for a deploy-anywhere approach, useful in the IoT domain.

An elastic system can also choose to deploy on both real-time enabled resources, for reliability, and on non-real-time enabled resources, for maximum average performance, simultaneously. As mentioned earlier, resiliency goes hand-in-hand with important features in the fourth generation of industry, such as on-line learning.

3.4 Control over the Cloud

Some distinctions concerning the architecture of the considered control systems is necessary. This will also introduce useful terminology. First, Control over the Cloud (CotC) refers to the situation in Figure 3.4. A distinguishing feature is that the control signal must leave the cloud to control an external plant. The logic that binds the plant to the cloud is referred to as the client. Exactly how this loop is closed is not defined. The input signal y can be read by the client through sensors on the plant but could also come from an external source. The client and the logic in the cloud together create the controller. Throughout the thesis, the plant is considered to be a physical

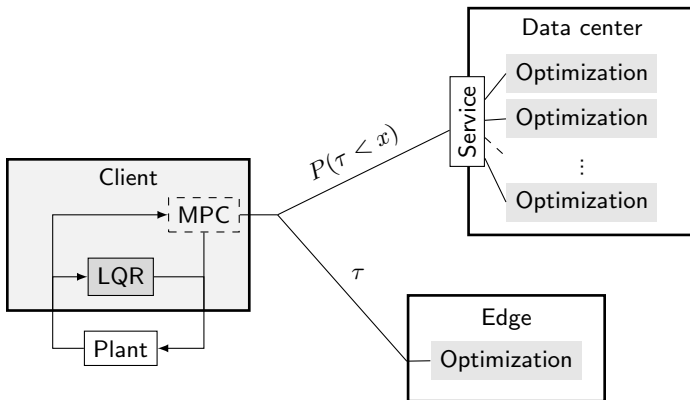


Figure 3.5 Cloud assisted MPC.

device and the closed loop is a cyber-physical system (CPS). The aim is to implement closed loop control in a CPS using a cloud control system (CCS), and more specifically a form of elastic control through utility computing.

Offloading

To implement the case in Figure 3.4, the client could be responsible only for connecting to the cloud and forwarding control input to the actuators. As mentioned when introducing offloading in Section 2.4, a client that can connect to the cloud is often also capable of implementing at least some control logic. We will therefore assume the general viewpoint that the client is offloading. In addition to its usefulness in practice, an offloading scheme serves as a good research platform. For example, with input and output in the same client, we do not have to be concerned with synchronizing clocks. The phrases *controller offloading*, the *controller in the cloud* and similar expressions are interchangeable. While we can argue that they should be differentiated because of differences in the implementations, as mentioned earlier, it should be possible to transfer most reasoning to either case.

Figure 3.5 provides an overview picture of the offloaded controller in a distributed cloud. The client contains a basic controller, the Linear Quadratic Regulator (LQR), and a virtual MPC, representing an advanced controller. The purpose of the MPC is to replace or *augment* the LQR so that performance is improved by using the cloud. The MPC can also be allowed to overrun its deadline and become infeasible, because there is a stabilizing controller on-board the client device.

In the setup of Figure 3.5, the MPC can execute many heavy optimizations in the data center but this comes with a large and unknown delay penalty. The edge cloud in Figure 3.5 lacks the abundance of resources and

can therefore only provide a single, less demanding optimization, but in return provides a fixed delay. In the distributed cloud infrastructure, the aim is to arbitrate controllers as necessary and achieve good, reliable performance in the long run. Keep in mind from Section 2.2 that while Figure 3.5 in many cases is the conceptual platform, the distributed cloud can offer more deployments. The cloud can also present the abstract user view in Figure 3.4, automating as many supporting systems as possible, including the physical and virtual location of computations.

4

Model Predictive Control

The strategy of Model Predictive Control (MPC), is to solve a finite-horizon optimal control problem in every sample. At each sampling instant, the controller optimizes a performance objective while satisfying (physical) system constraints. The explicit, on-line solution is capable of optimizing the system response with respect to the constraints, and the formulation of the optimal control problem is a fairly general control specification. Because the MPC has an intuitive design, applies directly to optimal control of multi-variable systems, and includes explicit specification of plant and actuator constraints, it has become a popular engineering tool. The primary draw-back is the need for extensive calculations to solve the problem on-line. A large part of the control engineering challenge is to find practical implementations and reduced complexity problems that can be solved in real-time. Linear system models, linear constraints, and quadratic cost functions are often used because solvers exist that can quickly provide optimal solutions.

The MPC strategy has been applied as a mature technology within the process industry for several decades. The survey [Qin and Badgwell, 2003] presents a classic setup, in which the MPC controllers are designed to drive the process from one constrained steady state to another, while minimizing constraint violations along the way. The example is of a processing plant with a hierarchical control system, and the conventional alternative that Qin and Badgwell describes is to use combinations of PID controllers, lead-lag blocks and selection logic, a method that is much less convenient than the MPC design. In this hierarchy, the MPC executes only once every minute and optimizes the configuration of a set of PID controllers, which work at a much higher frequency. As computing systems, optimization software, and MPC theory have evolved [Rawlings and Mayne, 2009b; Mattingley and Boyd, 2010; Bemporad et al., 2002], the online optimizing controller can now be used for complex control tasks also at very high frequency and applies control input directly to the plant rather than acting as a setpoint generator.

MPC was introduced as the method of choice to study controller offloading in Section 2.4. The use of wireless connectivity such as Fifth Generation

Wireless Specifications (5G) naturally sets the scope to applications in the single to two digits milliseconds response time domain. As will be shown in Part II, this is well aligned with practical lower limits for cloud control applications. This range is also suitable for a lot of automation in areas that now use MPC and have a vested interest in the fourth industrial evolution, such as factory automation [Qin and Badgwell, 2003; Vick et al., 2016; Maxim et al., 2019], the automotive industry [Hrovat et al., 2012] and aerial drones [Persson, 2019]. While approximations and explicit MPC [Bemporad et al., 2002] allow for a large range of MPC problems to be solved in real time inside on-board electronic control units, as detailed in [Hrovat et al., 2012], it remains a fundamental challenge that the problems can become too complex. Limited local computations (or other limiting resources such as memory and storage) is one reason to use the cloud, and it generates two research paths that need to be explored. One is that the cloud provides unique possibilities to extend the potential of a device. The second is the exploration of extensive, unknown computation times, in combination with a time-sensitive dynamic system and on-demand platforms.

In the following, the study of cloud controllers is based on standard methods that are easy to implement and extend. It is important to recognize that the implementations naturally extend to arbitrary non-linear specifications and non-convex problems. After reading Part III, the reader will also recognize that such solutions can co-exist with, execute next to, and gradually evolve from standard approximations, in a single application. For the method, it is interchangeable whether an extensive calculation comes from an inefficient solver, or longer and more iterations due to a complex specification, albeit the latter should provide better system performance. To provide the necessary background for the following text, and especially Part III, the following chapter presents an overview of the control strategy and the implementation used in the thesis.

4.1 Definition

The MPC solves the optimal control problem

$$\underset{\hat{\mathbf{x}}_k, \hat{\mathbf{u}}_k}{\text{minimize}} \quad \sum_{i=0}^{N-1} l(\hat{x}(k+i|k), \hat{u}(k+i|k)) + V_f(\hat{x}(k+N|k)) \quad (4.1a)$$

$$\text{subject to} \quad \hat{x}(k+i+1|k) = f(\hat{x}(k+i|k), \hat{u}(k+i|k)), \quad (4.1b)$$

$$g(\hat{x}(k+i|k)) \leq 0, \quad i = 0, \dots, N-1, \quad (4.1c)$$

$$h(\hat{u}(k+i|k)) \leq 0, \quad i = 0, \dots, N-1, \quad (4.1d)$$

$$\hat{x}(k+N|k) \in \mathcal{X}_f, \quad (4.1e)$$

$$\hat{x}(k|k) = x(k) \quad (4.1f)$$

in each sampling period T_s , provided an initial state $x(k) \in \mathbb{R}^n$. $\hat{\mathbf{u}}_k = \{\hat{u}(k|k), \dots, \hat{u}(k+N-1|k)\}$ is a sequence of optimal control inputs and $\hat{\mathbf{x}}_k = \{\hat{x}(k|k), \dots, \hat{x}(k+N|k)\}$ are the predicted plant states. From here on, bold notation and the short hand $x(i|k) = x(k+i|k)$ will be used for sequences of variables. N , specifying the number of predicted time steps, is the controller horizon. Here, a single horizon N is used for prediction and control. It is common to separate the MPC into one control horizon N_c and one prediction horizon N_p , where control can be applied only during N_c . With the exception of a special case in Chapter 12, a single horizon is used throughout the thesis. In Equation (4.1) the states $\hat{\mathbf{x}}_k$ are included as decision variables under minimize. A convention of not including them is used in later parts of the thesis.

Equation (4.1b) models the plant dynamics, (4.1c) defines state constraints with $g : \mathbb{R}^n \rightarrow \mathbb{R}^{p_x}$, and (4.1d) defines input constraints with $h : \mathbb{R}^m \rightarrow \mathbb{R}^{p_u}$. The set definition (4.1e), $\mathcal{X}_f \subset \mathbb{R}^n$, is referred to as the terminal set or terminal constraint. This constraint can be used to ensure that the problem enters a new constrained steady state at the end of the control horizon. Equation (4.1a) represents the cost-to-go as the value function

$$V(\mathbf{x}, \mathbf{u}) = \sum_{i=0}^{N-1} l(x_i, u_i) + V_f(x_N) \quad (4.2)$$

composed of the stage cost $l : \mathbb{R}^{n+m} \rightarrow \mathbb{R}_+$ and the terminal cost $V_f : \mathbb{R}^n \rightarrow \mathbb{R}_+$, both of which are decrescent, continuous and positive-definite functions.

DEFINITION 1—POSITIVE-DEFINITE FUNCTION

A real valued function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is positive-definite if it is non-negative, i.e., $f(x) \geq 0$ for all $x \in \mathbb{R}^n$, and $f(x) = 0$ if and only if $x = 0$. \square

The terminal cost V_f relates to the asymptotic properties of the controller.

The optimization variable is the control signal vector

$$\hat{\mathbf{u}}_k = \{\hat{u}(0|k), \hat{u}(1|k), \dots, \hat{u}(N-1|k)\}. \quad (4.3)$$

The solution provides a predicted optimal control sequence \mathbf{u}_k^* , a predicted path \mathbf{x}_k^* , and an optimal cost to go

$$V_k^* := V(\mathbf{x}_k^*, \mathbf{u}_k^*). \quad (4.4)$$

To implement feedback control, the implicit control law is

$$\kappa(k) = u_k^*(0). \quad (4.5)$$

Thus, the first control input in the optimal control sequence is applied to the plant and this process is repeated for $k+1$, i.e. the next sampling period T_s .

An MPC with soft constraints adds *slack variables* ϕ to the state cost of the value function (4.1a) to form

$$\sum_{i=0}^{N-1} l(x(i|k), u(i|k), \phi(i|k)) + V_f(x(N|k)). \quad (4.6)$$

The slack variables allows the controller to violate constraints at a high cost. They enter the definition by replacing (4.1c) with

$$g(x(i|k), \phi(i|k)) \leq 0, \quad \forall i \in \{0, \dots, N-1\}. \quad (4.7)$$

Without soft constraints, the optimizer can be faced with an infeasible problem due to discrepancies between the model and the actual plant dynamics, or unexpectedly large disturbances.

At times, the controller may erroneously decide that it cannot keep the system within constraints and conclude that the problem is infeasible, whereas a simple solution such as holding the current control action or applying the predicted solution (i.e. $u^*(k|k-1)$) may suffice to recover. Soft constraints provides a systematic approach that is often a better practical solution to ensure a feasible control problem [Levine and Raković, 2018; Maciejowski, 2002]. The downsides of soft constraints are the allowed (and possibly systematic) constraint violation, a larger optimal control problem, and loss of formal stability guarantees for open-loop unstable systems [Zeilinger et al., 2010]. Soft constraints are introduced into the control problem in Part IV. They are important in Chapter 14 in order to keep the remote controller feasible in the event of long delays.

4.2 Stability

Stability of the MPC is generally proven by showing that the value function is a Lyapunov function for the closed loop system, i.e., that

$$V^*(k) \leq V^*(k-1). \quad (4.8)$$

The deterministic nature of the time-invariant optimal control problem ensures that the infinite horizon nominal closed loop system

$$\underset{\mathbf{x}, \mathbf{u}}{\text{minimize}} \quad \sum_{i=0}^{\infty} l(x_i, u_i) \quad (4.9a)$$

$$\text{subject to} \quad x_{i+1} = f(x_i, u_i), \quad (4.9b)$$

$$g(x_i) \leq 0, \quad (4.9c)$$

$$h(u_i) \leq 0, \quad (4.9d)$$

$$x_0 = x(t) \quad (4.9e)$$

is stable and recursively feasible. For the finite-horizon problem, the terminal cost and terminal constraints are used to ensure stability. This section recalls some of the main stability results, presented in detail for instance in [Rawlings and Mayne, 2009a].

Consider the control problem in Equation (4.1) without the terminal constraint, i.e., replace (4.1e) with the constraint $g(\hat{x}(k + N|k)) \leq 0$, and no terminal cost, i.e., $V_f = 0$. Clearly, this controller is not concerned with what happens after its prediction horizon, i.e., after $k + N$. With a perfect model and no disturbances, the observed state at time k will correspond to the state predicted in the previous step, $\hat{x}(k|k) = \hat{x}(k|k - 1)$. Subsequent states however, can be different from the previous prediction because of the new time interval $\hat{x}(k + N|k)$, i.e.,

$$\{u(k|k - 1), \dots, u(k + N - 2|k - 1)\} \neq \{u(k|k), \dots, u(k + N - 2|k)\}, \quad (4.10)$$

and the cost can increase so that $V^*(k) > V^*(k - 1)$. With a sufficiently large N this problem is avoided and (4.8) will hold, but the necessary value for N can be difficult to find. A large N can also be intractable because of the larger problem that must be computed when increasing the value of N . Rather than using excessively large values of N and jeopardizing the stability of the controller, the terminal constraint and terminal cost is introduced to explicitly ensure stability of the nominal closed loop.

Consider an equilibrium at $x = 0$ and $u = 0$, i.e. $0 = f(0, 0)$, and the terminal constraint $\hat{x}(k + N|k) = 0$. The stability of this system is straight forward. Assume that $\hat{x}(k|k)$ is feasible and that \mathbf{u}_k^* is the optimal control sequence computed for $\hat{x}(k|k)$, and \mathbf{x}_k^* the corresponding predicted states. As a consequence of the equilibrium and the terminal constraint, $\mathbf{u}_{k+1}^* = \{u^*(k + 1|k), u^*(k + 2|k), \dots, 0\}$ is a feasible solution in the next iteration of the control loop. Using the notation that $x_k(i)$ indexes into a sequence so that $x_k(i) = x(k + i|k)$, the cost at time $k + 1$ is

$$\begin{aligned} V^*(k + 1) &= \sum_{i=0}^{N-1} l(x_{k+1}^*(i), u_{k+1}^*(i)) \\ &\leq \sum_{i=0}^{N-1} l(x_k^*(i), u_k^*(i)) - l(x_k^*(0), u_k^*(0)) \\ &\quad + l(x_{k+1}^*(N - 1), u_{k+1}^*(N - 1)) \\ &= V^*(k) - l(x_k^*(0), u_k^*(0)) + l(0, 0) \end{aligned} \quad (4.11)$$

as a consequence of the optimality of the solution \mathbf{u}_k^* and the ensured feasi-

bility of $u_{k+1}^*(N) = 0$. Rearranging and using telescope summation,

$$\begin{aligned} \sum_{k=0}^K l(x_k^*(0), u_k^*(0)) &\leq \sum_{k=0}^K V^*(k) - V^*(k+1) \\ &= V^*(0) - V^*(K+1) \\ &\leq V^*(0). \end{aligned} \tag{4.12}$$

Since l is non-negative, this leads to $l(x_k^*(0), u_k^*(0)) \rightarrow 0$ as $k \rightarrow \infty$, providing stability in the sense of Lyapunov. This however, depends on the feasibility of achieving $\hat{x}(k+N|k) = 0$, which is not practical. Thankfully, the results intuitively extend to a constraint set through the terminal constraints, \mathcal{X}_f , and terminal cost, V_f , which is how the stable controller is implemented. We consider this extension.

First, define the positive invariant set [Blanchini, 1999] of a control law such that if the initial state is in the set, the state will remain inside the set under the effect of the control law.

DEFINITION 2—POSITIVELY INVARIANT

The set X is positively invariant for the control law $\kappa(x)$ and plant function $f(x, u)$ if

$$x(0) \in X \Rightarrow x(k+1) = f(x(k), \kappa(x(k))) \in X, \forall k \geq 0.$$

□

Then introduce the invariant set definition of Lyapunov stability.

DEFINITION 3—LYAPUNOV FUNCTION

Let X contain a neighborhood of the origin in its interior and let X be a positively invariant set under the control law $\kappa(x)$ and the plant model $f(x, u)$. The (continuous) function $V : X \rightarrow \mathbb{R}_+$ is a Lyapunov function in X if for all $x \in X$ if:

$$\begin{aligned} V(0) &= 0, \\ V(x) &> 0 \quad \forall x \neq 0, \\ V(x(k+1)) &\leq V(x(k)) \end{aligned}$$

where $x(k+1) = f(x(k), \kappa(x(k)))$.

□

Also define the feasible set.

DEFINITION 4—FEASIBLE SET

The feasible set \mathbb{X}_N is the set of initial states that admit a feasible solution for the MPC problem with horizon N .

$$\mathbb{X}_N := \{x \mid \text{such that (4.1) is feasible and } x = x(k) \text{ in (4.1f)}\} \quad \square$$

THEOREM 4.1

Define a local control law κ_l and a terminal set \mathcal{X}_f such that \mathcal{X}_f is invariant under the control law κ_l . All state and input constraints must be satisfied in \mathcal{X}_f . Now assume that the terminal cost is a Lyapunov function in the terminal set \mathcal{X}_f , so that

$$V_f(x^+) - V_f(x) \leq -l(x, \kappa_l(x)) \quad \forall x \in \mathcal{X}_f \quad (4.13)$$

where x^+ is the next state following x , i.e., $x^+ = f(x, \kappa_l(x))$. The closed loop system under the implicit MPC control law (4.5) is stable and the system $x^+ = f(x, \kappa(k))$ is invariant in the feasible set \mathbb{X}_N . \square

Theorem 4.1 is proven by assuming a feasible state $\hat{x}(k|k)$ and an optimal control sequence $\mathbf{u}^*(k)$. The terminal constraint states that $\hat{x}(k+N|k) \in \mathcal{X}_f$ and the positive invariance that $\kappa_l(\hat{x}(k+N|k))$ is feasible in \mathcal{X}_f , and

$$x(k+1) = f(\hat{x}(k+N|k), \kappa_l(\hat{x}(k+N|k))) \in \mathcal{X}_f. \quad (4.14)$$

There is therefore always a feasible solution

$$\mathbf{u}^*(k+1) = \{u^*(k+1|k), u^*(k+2|k), \dots, \kappa_l(x^*(k+N|k))\} \quad (4.15)$$

in terms of the local control law. This solution, (4.15), represents a sub-optimal solution. Denote the terminal state of the sub-optimal solution $\bar{x}_{k+1}(N)$, and let $u_k^*(N) = \kappa_l(x^*(k+N|k))$ then

$$\begin{aligned} V^*(k+1) &\leq \sum_{i=1}^N l(x_k^*(i), u_k^*(i)) + V_f(\bar{x}_{k+1}(N)) \\ &= \sum_{i=0}^{N-1} l(x_k^*(i), u_k^*(i)) + V_f(x_k^*(N)) - V_f(x_k^*(N)) \\ &\quad - l(x_k^*(0), u_k^*(0)) + l(x_k^*(N), \kappa_l(x_k^*(N))) + V_f(\bar{x}_{k+1}(N)) \\ &= V^*(k) - l(x_k^*(0), u_k^*(0)) \\ &\quad + l(x_k^*(N), \kappa_l(x_k^*(N))) + V_f(\bar{x}_{k+1}(N)) - V_f(x_k^*(N)) \end{aligned} \quad (4.16)$$

and, because $V_f(x)$ is a Lyapunov function according to (4.13),

$$l(x_k^*(N), \kappa_l(x_k^*(N))) + V_f(\bar{x}_{k+1}(N)) - V_f(x_k^*(N)) \leq 0. \quad (4.17)$$

As a result, V^* is a Lyapunov function that can be used to show stability of the MPC system.

4.3 The basic MPC - limitations and extensions

The stability outlined in the previous section is pleasing in that it applies directly to the non-linear MPC, i.e., an MPC with non-linear dynamics and constraints, and arbitrary (continuous and positive definite) cost functions. It is also straight forward to extend to asymptotic stability. However, while a terminal set is a practical tool to ensure stability, the terminal constraints reduce the region of attraction of the MPC problem (4.1), i.e., the set of states that admit a feasible solution. Increasing N will enlarge the available state space but also increase computations. Another problem is the difficulty of finding a suitable terminal set and terminal cost. It is therefore common to not include the terminal set in the controller, and experimentally verify a horizon that is 'large enough' and can be computed on-line.

Robust control introduces random disturbances into the system dynamics and finds control actions that keeps the control problem feasible and stable, also in event of a sequence of worst case disturbances. Without robust control, the system dynamics are represented in state space form as

$$x(k+1) = Ax(k) + Bu(k) \quad (4.18)$$

while the robust controller introduces a random disturbance w into the equality constraint (4.1b),

$$x(k+1) = Ax(k) + Bu(k) + w(k). \quad (4.19)$$

To ensure stability and constraint satisfaction, the problem is translated into a conservative but deterministic form. In this thesis, robust control is not implemented. It is generally considered a direct extension.

Switched systems introduce time varying changes into the dynamics

$$x(k+1) = A_k x(k) + B_k u(k). \quad (4.20)$$

Unless these can be anticipated in (4.1b), the deterministic properties necessary in (4.16) are lost. The same goes for constraints and costs, but these values can often be anticipated, especially the latter, i.e., determining when to change the objective of the controller.

Finally, consider the constraint softening in (4.6) and (4.7). The recursively feasibility of the MPC is only assured for the nominal problem, and a small disturbance can cause it to become infeasible. Such disturbances always exist in practice and soft constraints are introduced to make the controller robust, ensuring that it remains feasible. One problem is that if a terminal set is used, which is required for stability, a disturbance can still cause the controller to become infeasible. Feasibility is ensured if the terminal constraints are removed but then the stability analysis no longer holds.

4.4 Linear and quadratic control

To implement the prototype controllers in the thesis, the standard method of a linearized model, box constraints (i.e. constraints on the form $x_{min} \leq x \leq x_{max}$) and a quadratic cost function is used. This control problem defines the stage cost

$$l(x, u) = x^T Q x + u^T R u \quad (4.21)$$

where $Q \in \mathbb{R}^{n \times n}$, $Q \geq 0$ is the state penalty and $R \in \mathbb{R}^{m \times m}$, $R > 0$ is the control penalty.

DEFINITION 5—POSITIVE DEFINITE MATRIX

The real valued matrix M is positive semidefinite ($M \geq 0$) if $z^T M z$ is positive or zero for every non-zero real column vector z . The matrix is positive definite ($M > 0$) if $z^T M z = 0$ only when $z = 0$. \square

The terminal cost is defined in terms of a terminal cost matrix Q_f ,

$$V_f(x) = x^T Q_f x, \quad (4.22)$$

and the linear constraints are

$$g(x) = C_x x - c_x, \quad C_x \in \mathbb{R}^{p_x \times n}, \quad c_x \in \mathbb{R}^{p_x}, \quad (4.23)$$

$$h(u) = C_u u - c_u, \quad C_u \in \mathbb{R}^{p_u \times n}, \quad c_u \in \mathbb{R}^{p_u}, \quad (4.24)$$

$$\mathcal{X}_f := \{x | C_f x - c_f \leq 0\}, \quad C_f \in \mathbb{R}^{p_f \times n}, \quad c_f \in \mathbb{R}^{p_f} \quad (4.25)$$

where p_x, p_u, p_f are, respectively, the number of state, inputs and terminal constraints. The plant model f is given in the standard state space form (4.18). Combining these components, the optimization problem becomes

$$\underset{\mathbf{u}}{\text{minimize}} \quad V(x_0, N) = \sum_{i=0}^{N-1} x_i^T Q x_i + u_i^T R u_i + x_N^T Q_f x_N, \quad (4.26a)$$

$$\text{subject to} \quad x_{i+1} = A x_i + B u_i, \quad (4.26b)$$

$$C_x x_i \leq c_x, \quad (4.26c)$$

$$C_u u_i \leq c_u, \quad (4.26d)$$

$$x_N \in \mathcal{X}_f. \quad (4.26e)$$

By joining x and u into the decision variable

$$\mathbf{z}(k) = [x^T(k+1|k) \quad \cdots \quad x^T(k+N|k) \\ u^T(k|k) \quad \cdots \quad u^T(k+N-1|k)]^T \in \mathbb{R}^{N(n+m)} \quad (4.27)$$

Terminal cost and terminal constraints

A convenient property of the QP problem is that it provides an easy solution to the problem of defining a terminal cost. This solution is found in the unconstrained, infinite-horizon, Linear Quadratic (LQ) problem. This standard problem is obtained by taking the cost function (4.21), the plant model (4.18), and an infinite horizon to form

$$\underset{\mathbf{u}}{\text{minimize}} \quad V(x_0) = \sum_{i=0}^{\infty} x_i^T Q x_i + u_i^T R u_i \quad (4.29a)$$

$$\text{subject to} \quad x_{i+1} = A x_i + B u_i. \quad (4.29b)$$

This form has a well known analytical solution providing a globally optimal solution. The cost-to-go of the closed loop optimal control problem is given by the solution to the discrete time Riccati equation

$$P = Q + A^T P A - A^T P B (R + B^T P B)^{-1} B^T P A. \quad (4.30)$$

Using the result from Equation (4.30) in Equation (4.22) provides the necessary Lyapunov function (4.13). Therefore, entering the result from (4.30) into H of (4.28) provides a known and stable controller action after the horizon. It is not ensured, however, that the constraints are satisfied. While it may often be sufficient to use a terminal cost in combination with a 'sufficient' horizon, stability is not formally guaranteed. As presented in Section 4.2, formal stability requires a terminal set that is an invariant set for the controller represented by the cost Q_f .

The control law arising from (4.30) is

$$\kappa(k) = K x(k) = -(R + B^T P B)^{-1} B^T P A x(k) \quad (4.31)$$

and it is possible to find a convex polytope, i.e., a set of p linear functions

$$\{x | F x \leq f\}, \quad F \in \mathbb{R}^{p \times n}, \quad f \in \mathbb{R}^p, \quad (4.32)$$

that provides a bounded invariant set of the closed loop

$$x(k+1) = A x(k) + B \kappa(k) = (A + B K) x(k) \quad (4.33)$$

if it exists.

An example is shown in Figure 4.1. This figure shows state constraints, a generated invariant set and two closed loop trajectories for a plant with two states and one input, regulated by an LQR. The dashed lines show the linear constraints of the polytope (4.32) that has been generated to form the invariant set. In addition to the state constraints that are shown in the figure, the invariant set also ensures that the controller does not violate the

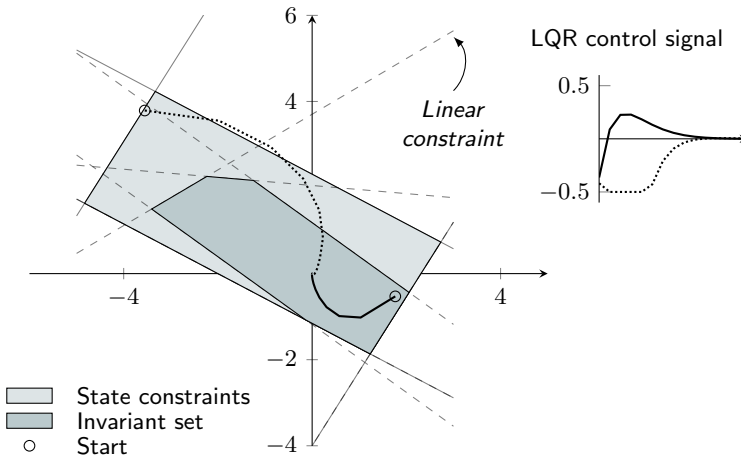


Figure 4.1 Example of invariant set for the closed loop system in Equation (4.33).

input constraint $|u| \leq 0.5$. Trajectories that start inside the invariant set will stay inside the invariant set. The solid trajectory is an example of this. The dotted trajectory, however, violates the state constraints and also saturates the control signal, as seen in the plot to the right in the figure.

While state constraints can often be 'soft' in the sense that they can be temporarily violated (they can be performance or comfort constraints), input constraints are often hard, physical constraints limited by the actuator. When the input constraints are hit, the linear properties of the system are no longer valid. The only safe way to avoid input saturation and constraint satisfaction is to ensure that Equation (4.33) is used inside the invariant set. There is however a much larger set of starting states that are controllable, i.e., for which there exists a sequence of control inputs that brings the state to the origin, without violating constraints. It is the job of the MPC to find these sequences.

Solvers

The QP is implemented in the thesis using various solvers: Matlab's [MATLAB, 2020] `quadprog`, QPgen [Giselsson, 2015] (using PyQPgen [Skarin, 2018]), Python CVXOPT [Andersen et al., 2021], and CVXGEN [Matingley and Boyd, 2012]. The latter three are used for optimizations that are performed in the cloud. While the calculation time of QPgen and CVXGEN are similar, CVXOPT is slower. One reason for this is that with QPgen and CVXGEN, the problem is specified offline to generate compiler optimized, machine native, dynamically loaded libraries, while CVXOPT is implemented

directly in Python. The latter is therefore more convenient and flexible. CVX-GEN is in turn more flexible than QPgen but also has downsides such as being limited in the size of the optimization problem and not providing open access. QPgen on the other hand is not easy to extend with custom MPC problems. No explicit benchmarks of the different solvers are provided since this is not the primary reasons for the selection. Rather, the ability to select a lower performing solver in order to benefit from other properties, is a feature that a powerful cloud can provide. As will be shown in Chapter 7, even the high performing solvers can run into problems on a relatively powerful IoT device.

5

Reference Plant

With a few exceptions, the ball and beam process in Figure 5.1 is used throughout the thesis to illustrate concepts and measure cloud performance. The objective of this plant is to position and balance a ball on a rotating beam. The controller sets the angular velocity of the beam and gets two measurements from the plant: the position of the ball on the beam, and the angle of the beam. The process has natural constraints in the length, angle, and speed of the beam.

While the ball and beam process is non-trivial, it is simple enough that it is certainly not necessary to use a powerful computer to control the plant. It is straight forward to implement a controller using non-demanding methods such as a LQR or a cascade structure of two PID controllers. It is also possible to implement an efficient explicit MPC on a very limited device to control the process. Nonetheless, this simple example is sufficient and useful for research. There are some basic benefits of using the plant: it is intuitive to evaluate, there exists a physical plant that is easy to implement for and verify, and its description can be compactly detailed. It is also good that the MPC problem can be solved without the most efficient solvers, and that, experiments can be designed to exhibit both short and long execution times. What is important, is that the controller works at a reasonable frequency in relation to the networks and computational demands. This is verified as part of the explorations in Part II.

The model in Figure 5.2 is sufficient to represent the process. This is the model implemented in Matlab Simulink and the base for simulations.

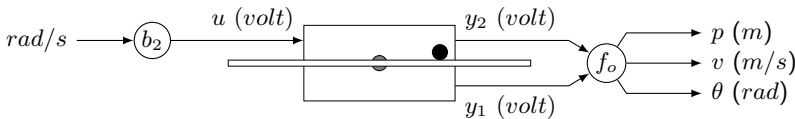
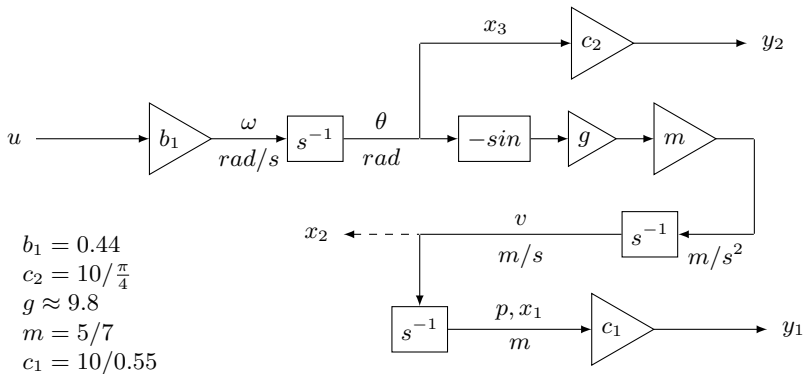


Figure 5.1 The ball and beam process


Figure 5.2 Ball and beam diagram

The model assumes that a change in the velocity of the beam is applied instantaneously and uses a simple non-linear acceleration of the ball without friction. The inputs and outputs, u and y , in this process are voltage levels, as shown in Figure 5.1. The state of the plant is represented by $x = [p \ v \ \theta]^T$ where p is the position of the ball, v the velocity of the ball and θ the angle of the beam. The function f_o in Figure 5.1 represents an observer that reconstructs and estimates state x from y . When used with the QP solvers, the model is linearized around $\theta = 0$.

The physical plant has three hard constraints, the beam length l , the angle θ_{max} and the input u . A box constraint is also created for the velocity. This represents a comfort constraint, but a large v_{max} can also cause model errors because of the linearized acceleration. The constraints are generally specified relative to the origin and are symmetric,

$$|x_1| \leq l/2, \quad |x_2| \leq v_{max}, \quad |x_3| \leq \theta_{max}, \quad |u| \leq u_{max} \quad (5.1)$$

More generally and represented as the inequality in (4.28),

$$\begin{aligned}
 C_x &= [I_3 \quad -I_3]^T, & c_x &= [x_{ub} \quad x_{lb}]^T \\
 C_u &= [1 \quad -1]^T, & c_u &= [u_{ub} \quad u_{lb}]^T
 \end{aligned} \quad (5.2)$$

where, relative to the origin,

$$\begin{aligned}
 x_{lb} &= [-l/2 \quad -v_{max} \quad -\theta_{max}], \\
 x_{ub} &= [l/2 \quad v_{max} \quad \theta_{max}], \\
 u_{lb} &= -u_{max}, \\
 u_{ub} &= u_{max}.
 \end{aligned} \quad (5.3)$$

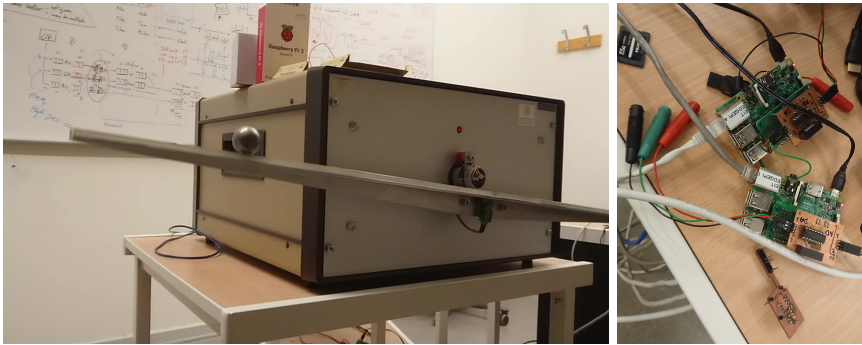


Figure 5.3 Left: The ball and beam. Right: Two client devices (Raspberry Pi) with a custom made analog-to-digital and digital-to-analog extension.

When implementing tracking, the origin is moved to the setpoint and these values must be updated accordingly.

The applicability of this basic process is verified in Chapter 7, after which simulations are used to provide reproducible results. The ball and beam plant that was used for the experiments in Chapter 7 is shown in Figure 5.3, next to the client device used to control it.

Part II

Cloud Performance

6

Introduction

The following two chapters present two research testbeds, with experimental results for mission critical control over the distributed edge cloud and cloud native offloading. The testbeds have evolved from the assumption, and investigated hypothesis, that critical feedback control systems can be built using commercial off-the-shelf (COTS) components and cloud services. The purpose of the experiments is to demonstrate running time sensitive and mission critical applications using different clouds and cloud native software, investigate feasibility, and find limitations of control over the cloud. The testbeds are used to measure the performance of current state-of-the-art, as an important first step towards frameworks that are capable of mission critical control over the cloud. This provides insights into the limitations of the systems described in Part III.

Chapter 7 looks at a research testbed that consists of a distributed set of compute nodes, a distributed PaaS framework, a 5G cell, and a time sensitive and mission critical process under control. The application executing on this testbed is a critical control system showing the potential for merged IoT, 5G and cloud. Mobility, reliability and low latency at the edge is provided through the 5G wireless radio. The control loop is implemented using flow-based programming [Szydło et al., 2017] on an IoT PaaS called Calvin [Persson and Angelsmark, 2015]. This environment provides the means to continuously execute a mission critical application, while seamlessly relocating computations to various geographically diverse locations.

Chapter 8 takes a closer look at cloud performance in an offloading scenario. Services that provide offloading are implemented using IaaS, Kubernetes and FaaS. This testbed is built for scalability and uses components that have become representative of cloud-native software. Chapter 8 extends the work in Chapter 7 with the offloading design, and an examination of performance loss from geographical distance and the use of high-level interfaces. It also takes on the question of how prominently *platform noise* from other tenants and *service logic* from the provider, appear in the delay profile towards a typical cloud.

6.1 Related work

Testbeds that join user equipments (UEs)¹, wired and wireless networks, distributed cloud infrastructure and platforms are crucial instruments to realize and study the complexity of the edge cloud. The literature contains a number of such attempts. The authors of [Kang et al., 2013] present the SAVI testbed. SAVI is an edge cloud testbed realizing Network Function Virtualization (NFV) with a Field-Programmable Gate Array (FPGA)-cloud. SAVI is used in the investigation of virtualization of the wireless access network. Similarly, in [Wan et al., 2016], a full testbed using existing wireless technologies, IoT frameworks, and devices is deployed on an actual production line. In [Hu et al., 2016] the authors implement a rudimentary edge cloud testbed to quantify the impact of edge computing on mobile applications using WiFi and the public 4G network. Their effort reveals significant latency and energy usage improvements compared to distant Data Centers (DCs).

[Tärneberg et al., 2016] studied the performance of cloud native applications on commercially available platforms in a smart city context. The study implemented an IoT infrastructure for smart traffic lights using a standard message passing framework, function services, and stateful storage, all provided as cloud services. The response time with this scalable IoT design, from sensed events to control actions, was on average over one second and also involved long tail times.

[Leitner and Cito, 2016] provides a benchmark of IaaS from several public cloud providers. The goal of this study is to investigate how accurately the performance of an acquired virtual machine can be estimated in advance. Results showed substantial performance differences between cloud providers and different regions. No substantial impact was seen from hardware heterogeneity in a region, or performance variations correlated with the time of day. It was concluded that multi-tenancy is an important factor and that selection of instance based on cost is non-trivial. The authors point out that cloud performance is a moving target, making it important to continue with benchmarks, and that users perform careful evaluations.

Other works have attempted to characterize and profile different aspects of the edge cloud. Tärneberg et al. have developed simulators for studying the dynamics of the edge cloud, culminating in [Tärneberg et al., 2017b], a paper that provides a structure for resource management in the distributed cloud. The simulation software iFogSim [Gupta et al., 2017] offers a platform for conceptual exploration of resource management techniques in the IoT and edge cloud environments, through simulation.

The authors of [Mahmud et al., 2014] evaluate a generic platform for in-

¹A device used by an end-user to communicate in a mobile broadband network, such as phones and IoT devices

dustrial control, with respect to latency, jitter, throughput, and CPU load. Their focus is on the pros and cons of virtualization in a multi-core environment rather than the cloud. Similarly, [Horn and Krüger, 2016] implements a water tank control process and evaluates latency over a virtual Software Programmable Logic Controllers (vSoftPLC) on top of LinuxRT. In both of these works, the system implementation is evaluated, rather than control of the plant. In [Hegazy and Hefeeda, 2015], *Industrial automation as a cloud service* is proposed. The authors evaluate a system of time sensitive control processes in a one-tier distributed cloud environment. Latency compensation is modeled and redundancy with stability and smooth controller handover is achieved.

The application testbed in Chapter 7 is distributed, event-driven, serverless, and based on a data-flow programming model. There are a number of such platforms for different workloads, e.g., Nebula [Ryden et al., 2014], Node-RED [OpenJS Foundation and Node-RED contributors, 2021], IEC 61499 [Vyatkin, 2011], and Naiad [Murray et al., 2013]. These systems are targeted for the IoT domain and cater for workloads varying from simple event-driven automation to high-throughput Hadoop² jobs. Calvin, which is used in Chapter 7, is most similar to Node-RED. However, while Node-RED emphasizes the programming model and graphical tools, Calvin puts more focus on runtime dynamics and distributed deployment. The perspective of Calvin is well attuned to the presentation of the distributed data-flow model in [Giang et al., 2015] where a Distributed-NodeRED (DNR) extension is proposed. A notable operational difference is that DNR employs duplication to realize mobility, while Calvin implements the code migration technique, which is arguably more efficient and is better suited for computationally intense applications [Giang et al., 2015]. Additionally, Calvin can migrate work loads using various performance criteria, to provide, for instance, load balancing or jitter reduction.

The interest in using information and communication technology (ICT) as a means to implement networked control using COTS predates the cloud era. An example is found in [Kim et al., 2006], where control is implemented over two private Local Area Networks (LANs), to balance a steel ball in a magnetic levitation system. The engineering merit of COTS, which at that time consisted of the Internet, general purpose networks, a web server, and the Common Gateway Interface (CGI), is explicitly acknowledged in this work. More recently, modern systems such as drones have been considered [Pelle et al., 2019], and the potential for implementation using high levels of abstraction.

Research on deploying traditional feedback control loops to a cloud in general, is often focused on making cloud computing platforms and interme-

²Apache Hadoop, a software framework for distributed big data processing

diated networks behave as real-time systems. Targeted efforts have, for example, been made in the areas of deterministic dynamic networks [Gupta and Chow, 2008], resource allocation in data centers [Ahn and Cheng, 2015], and feedback control implemented in the network [Rüth et al., 2018]. An early example of successful attempts at deploying feedback control loops that span a cloud VM and back is found in [Esen et al., 2015]. The work in [Esen et al., 2015] identified that time-varying network delays is a key challenge, and it was shown that the availability gap introduced by the cloud could be spanned reliably across multiple clouds, given comprehensive middle-ware. With more consideration to control theory, the work in [Kaneko and Ito, 2016] use redundant PID controllers and Smith prediction to create a fail-over system, that is both resilient to feedback controller and network failures. On a tangent line of research, the authors of [Vick et al., 2016] demonstrate that the delay incurred by the cloud and the intermediate network can be accommodated for by designing the feedback control loop for a static delay and using open loop prediction. Also, not to be underestimated, security is a principal concern when operating in the cloud. On this topic, the authors of [Alexandru et al., 2018] showed that the performance overhead of security was manageable in a cloud-deployed feedback controller.

6.2 Research gap

Most systems surveyed in the related works are not mission critical and time sensitive at the scale and complexity addressed in the following work. The SAVI testbed is comprehensive, but does not provide a general edge cloud implementation for cloud native applications, nor does it span from the device across multiple tiers of cloud resources. The industrial applications targeted in [Wan et al., 2016], and other works, focus on framework integration rather than system and application performance. Also, they are not time sensitive.

Simulation tools are valuable, but cannot replace prototypes that capture the true complexities of the final applications. The experimental evaluations in the following chapters are justified, because a testbed could be practically implemented, avoiding assumptions and allowing investigations of the complexities of real clouds and networks. The testbed and application in Chapter 7 are also unique in its combination of a time sensitive control loop, an edge cloud infrastructure, massive MIMO wireless technology, and a PaaS with migration support. It was constructed to examine the relatively unexplored hypothesis, that the above combination will make it possible to implement critical systems using wireless connectivity, COTS components, on-demand computing, and flexible deployments. No PaaS similar to Calvin had been experimentally evaluated for critical control of CPSs at rates that challenge the low latency aspect of 5G. Online software migration is also a

new study in this context. The potentials of deploying software anywhere on a heterogeneous, spatially and interactively diverse cluster, then migrating calculations to the best location had been suggested but not explored.

This extends also to the basic system performance explored in Chapter 8. Works that study offloading in ordinary clouds, and control over the Internet, often focus on bench-marking the relative performance of similar tools, the services of a single cloud provider, and chains of execution, similar to the application in Chapter 7. A large portion of current state-of-the-art propose to change the notion of the cloud in order to fit the needs of feedback control systems, and are not pursuing an adaptation of feedback control systems to the reality of the cloud. There is limited focus on the control challenge and the basic performance of clouds and networks in general. In the broader community, delays and uncertainty in the order of hundreds of milliseconds are often assumed and studied. This is a consequence of chains of microservices and other bottlenecks in the applications. Individual service requests and stateless execution can achieve much better results. There is important progress to be made by considering and experimenting on unmodified clouds, when investigating boundaries and possibilities in utility computing.

7

Towards Mission Critical Control

The Wide Area Networks (WANs) separating a time sensitive and mission critical system from a traditional distant DC, often incurs a variable delay beyond what is operationally acceptable [Schulz et al., 2017]. The edge cloud was proposed to mitigate the latent latency, jitter, throughput, and availability barriers that separate the end users from distant DCs. To take advantage of the edge cloud, applications should adopt contemporary software platforms, such as PaaS. The following chapter presents a testbed for time critical control systems. The testbed is complete in that it includes ultra reliable wireless communication, IoT devices, edge compute connected to the access network, and connections to data centers. A distributed PaaS allows an application seamless access to the complete infrastructure.

7.1 5G enabled testbed

Historically, control systems have been deployed as monolithic software implementations on carefully tuned hardware, adjacent to the plants they control. Deploying monolithic software on static hardware makes such systems undesirably non-modular, less extensible, and limits their ability to self-adapt. Conversely, cloud-native applications are built for the cloud with the prospect of widening and deepening the penetration of cloud resources through the IoT and the edge, offering the prospect of greater flexibility, reuse, availability, and reliability with lower latency and jitter. When applications are implemented as loosely-coupled components, such as microservices [Dragoni et al., 2017], their execution can be distributed across the system's many nodes, migrated, and scaled to meet their individual objectives, as well as that of the system as a whole. To adapt to, and prosper in,

This chapter is based on [Skarin et al., 2018]

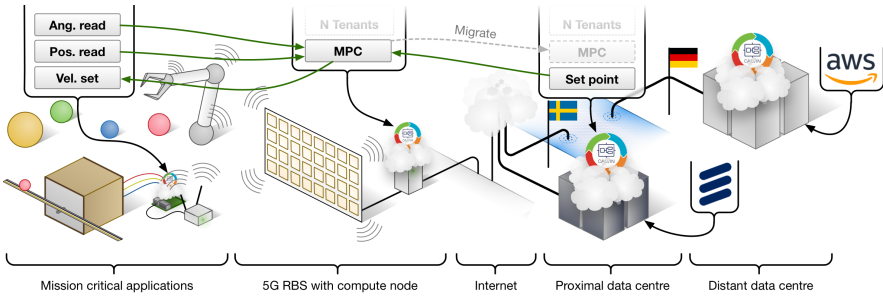


Figure 7.1 A flow programming implementation of critical control, deployed in runtimes on different machinery in several geographical locations.

the edge cloud, applications will arguably have to adhere to a cloud-native paradigm.

When accessing the edge cloud over URLLC 5G [Shafi et al., 2017], the delay and jitter are sufficiently low that time sensitive and mission critical applications can be deployed in the edge cloud. An edge cloud can also provide an application with redundancy and fall-back solutions at various geographical points in the infrastructure, for additional resilience. Deploying mission critical applications over the edge cloud must arguably occur in conjunction with the availability of edge cloud resources, the flexibility of cloud-native applications, and the reliability and low latency of next generation communication systems. There are many challenges and performance uncertainties in this premise. Therefore, we study the feasibility of deploying time sensitive and mission critical applications, and their performance, when deployed on an actual edge cloud infrastructure.

7.2 Edge cloud and massive MIMO

The research testbed, shown in Figure 7.1, consists of:

- 1) A radio *transmitter* and a *receiver*¹ (UE) constituting a 5G cell.
- 2) PC-type *compute nodes* in DCs and adjacent to the radio transmitter.
- 3) A reduced capacity *input-output* device and a *physical plant* at the radio receiver end.

Here, the physical plant can be a mechanical device that continuously performs a task, for example a robotic arm or an autonomous vehicle. The process that controls the plant is mission critical and time sensitive. The radio

¹More simultaneous receivers (and device clients) are supported by the testbed but only one is shown.

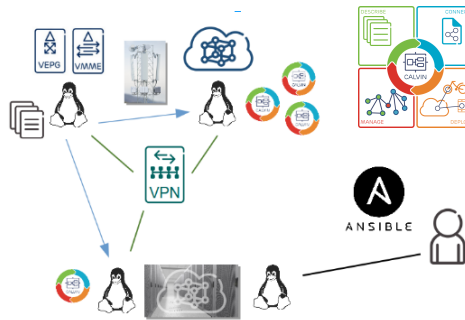


Figure 7.2 Illustration of a prototype deployment of the system in Figure 1.2 at WASP WARA-CAT in 2017. The base station, located outside of Gothenburg, Sweden, provided virtual machines and access to the virtualized 5G core. A distributed software platform was created using Calvin and a data center in Lund, Sweden.

subsystem is capable of parallel, synchronized low latency communication with several receivers.

The target is a platform with enough knowledge about the application to perform load balancing while allowing an application its own mobility. Further, there is a strong interplay between the edge cloud and the end user equipment, such that an application can automatically scale on top of the cloud and provide fall back on local devices. In the remainder of this chapter, the research testbed is referred to as *the system*.

Fifth generation wireless specification

A 5G wireless system represents the next generation wireless infrastructure [Shafi et al., 2017]. The emerging focus of 5G is URLLC and Massive Machine Type Communication (mMTC), where a large number of IoT devices, can reliably be served simultaneously at a low latency, $\leq 5ms$. These conditions cannot be replicated with current 802.11 (Wi-Fi) or Long Term Evolution (LTE) systems. A next generation wireless network also implies a deeper integration with associated cloud computing resources. On-demand resources are integrated into the radio base station (RBS) and access networks, to off-load the back-haul and eliminate the latency overhead of traversing multiple networks and providers.

The 5G network can service devices with very different requirements by deploying services on elastic clouds at various locations in the infrastructure. Figure 1.2 showed an illustration of such a system, from an Ericsson white paper published in 2017 [Ericsson, 2017]. Figure 7.2 shows an early independent prototype that was setup at Asta Zero testing grounds, as part of the



Figure 7.3 The LuMaMi massive MIMO [Malkowsky et al., 2017]

WASP WARA-CAT program in 2017. It connected virtualized resources in a base station with a data center cloud and implemented a Calvin PaaS. This prototype is similar to the testbed detailed below, but was not used to access a wireless interface or control an actual plant.

Massive multiple-input-multiple-output (MIMO) is an emerging radio access technology (RAT) for 5G and beyond. Fundamentally, massive MIMO is a multi-user MIMO (MU-MIMO) scheme, which can simultaneously communicate with multiple UEs on the same wireless resource. On the RBS-side, massive MIMO is typically configured with an order of magnitude more antennas than simultaneously served UEs. This is significantly more antennas than existing LTE-based RATs. Consequently, the system's spectral efficiency is a few orders of magnitude greater than existing RATs. The increased spectral efficiency can be used towards serving more simultaneous UEs, increase throughput, or realizing high reliability, beyond what can be achieved with existing RATs.

The testbed implements wireless access technology using the Lund Massive MIMO (LuMaMi) (Figure 7.3). LuMaMi is a massive MIMO testbed that was built using COTS components at Lund University, Sweden. LuMaMi's scope and detailed implementation are found in [Malkowsky et al., 2017]. LuMaMi can be seen as one solitary 5G cell that can simultaneously communicate with twelve UEs. Using LuMaMi, traffic can be routed directly through the system at the medium access control (MAC) layer, allowing the placement of a compute node in the RBS. This is referred to as an RBS break-out.

LuMaMi is configured according to [Tärneberg et al., 2017a]. The modulation scheme and measured performance are shown Table 7.1 next to typical, theoretical, 4G figures. While the achieved throughput may seem low, it is on par or better than contemporary 3GPP machine type communication standards. It is also more than sufficient to support the application. While LuMaMi can provide record-setting throughput [NI, 2016], the used config-

Table 7.1 Configuration with measured values for LuMaMi and the theoretical 4G performance of a public ISP.

Access	LuMaMi	ISP
Radio Technology	Massive-MIMO	LTE Cat13
Modulation	QPSK	Up to 64-QAM
Uplink	4.6 Mbps	75 Mbps
Downlink	9.1 Mbps	300 Mbps
Reliability	Ultra-reliable	Low
Minimal latency	5 ms	10 ms

Table 7.2 Node types

Node	Device	Location
Plant	Raspberry Pi 3B	Plant adjacent
RBS	Intel Core i7 Desktop	LuMaMi adjacent
ERDC	Intel Core i7 VM	Lund, Sweden.
AWS	Intel Xeon VM	Frankfurt, Germany

uration premieres a low latency and a highly reliable wireless link.

Edge cloud and network

The system as a whole is tied together by a set of compute nodes joined by a network. A summary of the compute nodes is presented in Table 7.2. Adjacent to the plant is a Raspberry Pi. In order to sample and manipulate the plant, the Raspberry Pi has been equipped with a ADC/DAC shield. The device also serves as a compute node and may host software in addition to the software required for interacting with the plant. The Raspberry Pi is also connected to a 5G UE. The 5G cell is isolated in its own sub-net. The sub-net includes the wireless infrastructure, an edge cloud node, and plant nodes. The plant-adjacent Raspberry Pis are connected to the system's sub-net over LuMaMi. The wireless edge cloud node is adjacent to the LuMaMi RBS. It connects directly to the RBS without traversing additional networks.

A router is located between the cell's sub-net and the larger DCs. The Ericsson Research Data Center (ERDC) resides in Lund, Sweden a few kilometers from the cell. ERDC is a research DC operated by Ericsson, that is open to industrial and academic research efforts within the Wallenberg AI, Autonomous Systems and Software Program (WASP). When the experiments were made, the cloud management platform in ERDC was Open Stack Pike and the servicing VM (a c4m16) has four Intel i7 cores registered by Linux as 1.6 Ghz CPUs and 16 GB of RAM. The Amazon Web Services (AWS) EC2 instance (a c4.large) is hosted on eu-central-1 (Frankfurt,

Germany). This node has two Intel Xeon cores at 2.9 Ghz and 8 GB of RAM. The application does not make use of all CPU cores. The two VMs on ERDC and AWS connect to the sub-net over a Virtual Private Network (VPN), allowing direct access between all compute nodes.

7.3 Cloud native application

On the investigated platform, a cloud-native IoT application is defined to be an application that has been dis-aggregated into logical and independent components connected in a *data-flow graph*, and that is hosted on a PaaS framework. The PaaS and its resident applications ubiquitously operate over multiple geographically distributed and heterogeneous compute nodes. An application's data flow graph can be rerouted and extended at run-time, when for example adding a new feature. Additionally, the components are able to traverse the cloud and associate with, and discover, physical input-output devices, if the application so requires.

Calvin

The testbed runs Calvin, a PaaS aimed at merging IoT and cloud in a unified programming model [Persson and Angelsmark, 2015]. Calvin is conceptually structured as follows.

Actors The operational units of Calvin are called actors (nodes in data-flow). These are implemented in Python by the user.

Runtimes² A runtime is an instance of the Calvin application environment on a device.

Tokens An actors' input and output messages, are known as tokens. The actions of an actor are triggered by the arrival of tokens on its input queues, and the result of an operation, a *production*, is placed in output queues. Each runtime independently schedules its resident actors in a round-robin manner.

Migration and scaling Actor states can be migrated and horizontally scaled across nodes. What constitutes an actors' state is defined by the developer. The Calvin framework can autonomously migrate and place actors to meet performance goals. Application owners can also specify requirements for actors which tie them to a preferred runtime. For example, a sensor reading actor can be tied the node associated with the physical plant it is meant to observe.

²In the implementation there is a one-to-one mapping between Calvin runtimes and compute nodes, they are therefore interchangeably referred to simply as nodes.

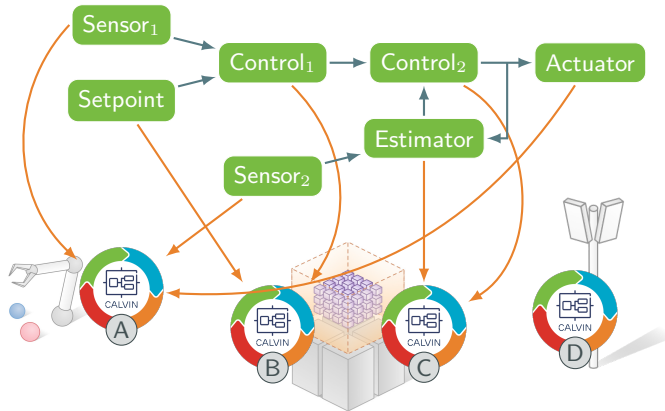


Figure 7.4 Example of cascade control Calvin application deployed on a set of runtimes. The green boxes are Calvin actors, the circles represent runtimes. No actor is initially deployed on runtime D.

Application Calvin applications are described in terms of reusable actor components. A set of actors and their interconnections constitute an *application*. The *CalvinScript* defines a simple language used to connect actors to form a directed graph, creating a Calvin application.

Distributed execution The application is deployed to a mesh network of distributed Calvin runtimes, as illustrated in Figure 7.4. The location of an actor and how its data is transported to other actors, is handled dynamically during deployment and throughout the application lifetime. The set of runtimes form a distributed execution environment. This environment manages actors and allows updates of the deployment, duplication of actors, and actor migration. Figure 7.5 illustrates the distributed execution and the overlay transport network provided by the PaaS. This figure also shows that $Control_2$ has migrated from runtime C, where it was deployed in Figure 7.4, to runtime D.

A Calvin developer implements actors that are responsible for isolated tasks. An actor should respond to messages on its (queued) input ports with messages on its (queued) output ports. The actions of the actor should not have side effects - inputs such as reading a sensor should be provided by other actors, dedicated to that task. Actors are implemented using a specialized Python library. Through Python decorators, the library creates an environment that triggers actor functions based on conditions on the actor's in-ports, and allows the actor to place productions (tokens) on its out-ports.

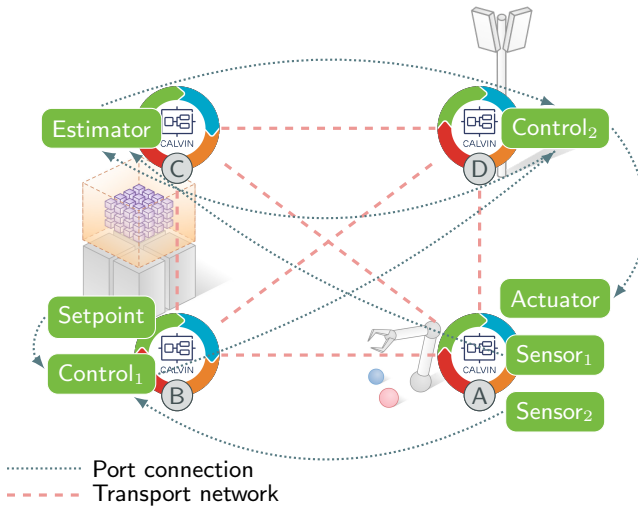


Figure 7.5 The Calvin application in Figure 7.4 executing in its distributed environment. Notice that Control₂ has migrated to runtime D from its initial deployment on runtime C.

The runtime environment takes care of executing the actors, transferring the token from out-ports to in-ports, handling token queues, duplicating tokens to separate branches, moving an actor to a new runtime, enforcing deployment requirements, meeting QoS etc.

A Calvin script specifies relationships between actors, options for automation such as scaling, and actor affinities (optional device requirements) using the CalvinScript language. The library of actors is made available to the cluster of runtimes and the application definition, the Calvin script, can be deployed to any of the runtimes. The actor affinities ensure that an actor such as Sensor₁, Sensor₂ and the Actuator in Figure 7.5 are deployed on a runtime with access to the input and output devices. Other actors, such as Estimation and Control can be located on any runtime. These can be *migrated* freely to obtain access to, for instance, low latency communication or more CPU time. The Calvin runtime system itself uses a distributed key-value store to manage actors and the distributed runtime environment.

7.4 Implementation

Control application

The plant under control in the experiments is the ball and beam process from Chapter 5. An optimizing controller is implemented in Calvin and used to control the plant. For the optimization, a dynamically linked binary is created with the use of QPgen [Giselsson, 2015]. The implementation uses a sampling period of 50 ms (20 Hz). This choice is a reasonable trade off between control performance and early observations of the system’s latencies. QPgen is an efficient solver and the optimization problem has few variables, yet we shall see that the time it takes to find a solution can be considerable. Sometimes there is no solution, or one is very hard to find. In such case the search ends after a fixed amount of optimization iterations.

The Calvin application graph for the MPC control loop is shown in Figure 7.6. The rounded rectangles represent individual components, implemented as actors, which are deployed onto the systems. To handle the presence of plant and sensory noise, and to reconstruct the full plant state, including the velocity of the ball, a Kalman filter is included. This could reasonably be its own actor, but here it is implemented in the MPC actor. The two Analog to Digital Converter (ADC) actors adhere to component reuse and the principle idea that they need not be collocated. However, they are to be read jointly and therefore share a clock tick. The setpoint actor is responsible for periodically (every N_s seconds) generating a new setpoint for the position of the ball. The components within the gray area can be freely placed within the system. The ADCs (the position and angle sensors) and the digital to analog converter (DAC) (the motor actuator) have an affinity to the plant-adjacent node.

A nominal controller is deemed useful to study the performance of the platform. Sensor input must be filtered, but the controller does not account for delays.

Execution properties

All related software runs on top of Linux. The Calvin runtime is launched using the Linux real-time scheduler with the POSIX FIFO scheduling policy. The edge node at the RBS can take full advantage of the real-time configuration³ as it runs directly on the physical hardware. In contrast, the guest operating systems hosting Calvin runtimes in the cloud, can be arbitrarily interrupted. The details of this depends on the cloud provider. The hosts operating systems may very well use the ordinary Linux Completely Fair Scheduler (CFS) to manage several, more or less active, virtual machines on the same host. A heavily loaded host may start a virtual machine migration

³Using vanilla real-time support, the kernel is not preemptive [Rostedt and Hart, 2007]

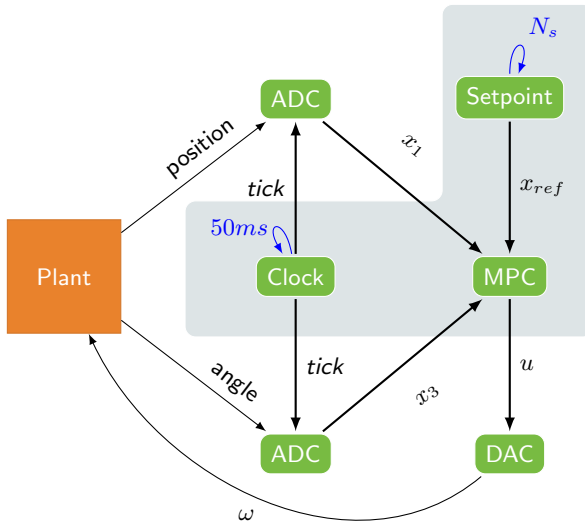


Figure 7.6 Calvin MPC implementation.

(Section 2.1) that slows down the system. There is no easy way to know or configure this. We can only assume that the scheduling and inter-node communication in the software platform will introduce a significant amount of delay and jitter, with a negative effect on control performance.

7.5 Testbed evaluation

The testbed performance is evaluated in a set of experiments designed to:

- 1) Reveal characteristics by establishing a baseline observation of the performance and behavior of the controller, over long time periods.
- 2) Verify the adaptability of the system by, at run-time, continuously migrating the controller actor across the system's nodes.
- 3) Measure control performance and execution properties to observe the advantages and disadvantages of different placement of the controller.

In order to observe the system's performance potential, the study is limited to normal operating conditions. Networks and clouds are shared and the connections to the DCs may at times degrade. Significant changes in the network and cloud behavior are assumed to be infrequent transient behaviors and are not handled specifically in this work.

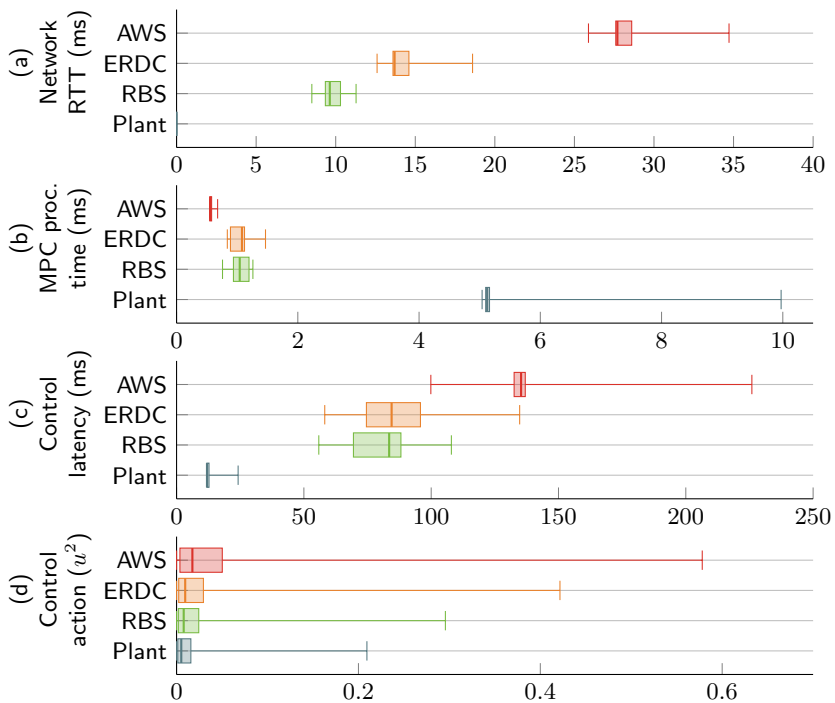


Figure 7.7 Statistical summary of the MPC baseline measurements. A box shows the (lower) 0.25-quartile to the (upper) 0.75-quartile. The line inside the box is the median. Whiskers show the lower quartile $- 1.5 \cdot IQR$ and upper quartile $+ 1.5 \cdot IQR$, where IQR is the interquartile range (the difference between upper and lower quartiles). Outliers beyond the whiskers are excluded.

System characteristics

To characterize and verify the basic functionality of the system, the MPC is executed on each of the nodes in Table 7.2. With each test, the MPC controls the beam for 60 minutes, while periodically changing the requested position of the ball, alternating between the center position and one side of the beam. To be robust in this experiment, the setpoint is restricted with a large margin to the end of the beam. The further out on the beam the ball moves, the more the controller is put to work. This is due to active constraints, which is return to in a later section.

Figure 7.7a shows the network round trip times (RTTs) from the Raspberry Pi at the plant to the other systems. Notably, the wireless link realized with LuMaMi introduces a latency of $5ms$ one way, as made evident by the

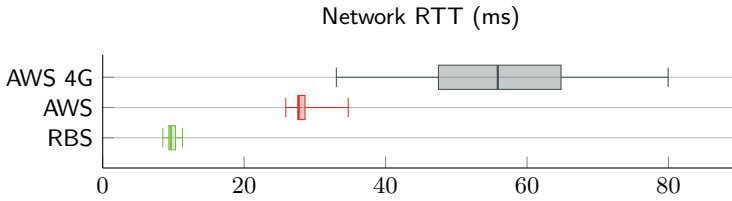


Figure 7.8 A comparison of the round-trip-times through LuMaMi and over a public access network using LTE. The configuration of the boxes are the same as in Figure 7.7

10ms RTT between the plant and the RBS node. 5G is pushing for even faster RTT, but this is good radio link performance compared to commercially available alternatives. To show how effective the system is in providing low latency access, Figure 7.8 compares the measured run-trip times to a commercial LTE network. The AWS 4G in this figure is accessing AWS over a public Internet Service Provider (ISP) using an ordinary LTE modem. The measurements for AWS and RBS in Figure 7.8 are the same as in 7.7a.

Figure 7.7b shows the MPC processing time. A simple scenario is chosen, where all nodes can execute the MPC with a significant margin. However, the Raspberry Pi at the plant is many times slower than the other systems. The AWS node is faster than RBS and ERDC, which is to be expected by the specification in Section 7.2. We see in this graph that there are large outliers in terms of processing time at the plant. Even though the MPC executes in real-time mode on the Raspberry Pi, recurrent extended system interruptions have been observed on the device. This could be the cause of these outliers. On the DCs, real-time properties do not apply outside the virtual machine, as far as known, and therefore some outliers are expected. Due to the short processing times, however, they are expected to be rare, which is also observed.

Figure 7.7c shows the full latency from reading the position of the ball to applying the control signal (i.e. adjusting the velocity of the beam). This is an important measure, since the controller is designed with the assumption that the input to output is instantaneous and hence, that the state of the system has not changed when the control signal is applied. The differences in delay are not as pronounced in Figure 7.7c as in Figure 7.7a. As seen from the magnitude of delays in Figure 7.7c, processing times in Figure 7.7b and the network latency in Figure 7.7a are far from the only contributors to the control latency. This tells us that a significant proportion of the delay in our system is introduced by the software platform or application design, and not the network. In contrast, results have shown up to 90 percent of the delay in public LTE (4G) systems is caused by the cellular network [Schulz

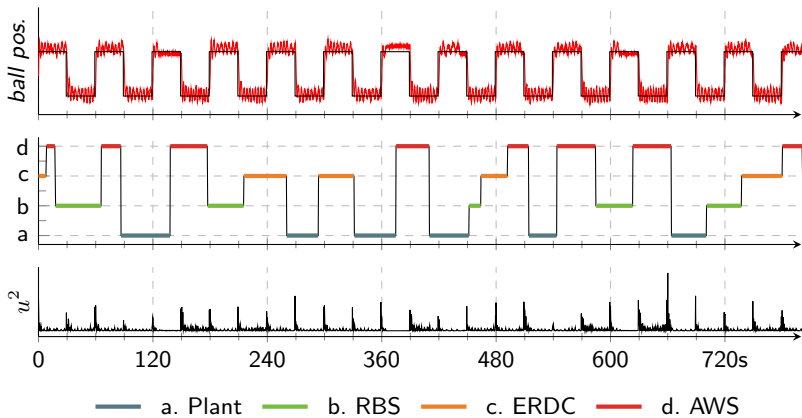


Figure 7.9 Time-series of MPC being randomly migrated between the system’s four nodes located in the remote data center (AWS), the municipal data center (ERDC), at the radio base station (RBS) and in a plant-adjacent Raspberry Pi (Plant).

et al., 2017], i.e., a very different situation. The effect on our process due to these properties are visualized in Figure 7.7d, where we see that the power of the control signal u increases the further the MPC is moved from the plant. Notice that the AWS node performs well on average, but network delays causes it to exhibit a larger mean and variance in the control signal. Such an effect can be part of the heuristics when deciding where to place control in the edge cloud.

System adaptability

It has been established that a controller can successfully be implemented on the edge cloud testbed, and characteristics in terms of processing times, latency, and jitter, have been investigated. Essential to the flexibility of the system is its ability to migrate applications and actors to respond to the application’s, and the infrastructure’s changing objectives. During a migration, the Calvin cluster performs the necessary modification of the network communication path, recreates the actor at the target node, copies state, and handles the transition of the token queues. Although the actor moves point-to-point, changing the communication paths may involve many nodes in the cluster. This perspective is now added by dynamically relocating the MPC among the nodes, while balancing and repositioning the ball.

In Figure 7.9, the MPC actor is continuously migrated randomly across the four compute nodes, at run-time. When doing this, the system must ensure to keep the Kalman filter, the setpoint, the previous state, and tracing

meta data intact. Delays, data loss or duplication, and incorrect state transfer negatively impacts the control performance. The plot in the middle of the figure shows the placement of the actor in time. The plot at the top shows the setpoint and the registered position of the ball. The bottom plot shows the output from the controller, again as the instantaneous power of the applied control signal, i.e., the square of the control signal.

Figure 7.9 shows that the process is stable and is able to operate without interruptions. The ball stays on the beam and close to the desired position. The upper plot confirms what is presented in Figure 7.7d, i.e., the control signal increases as a function of the distance to the plant. Setpoint changes are clearly visible as peaks in the control signal, but the migrations in the middle plot are not evident from the control signal nor in the ball's position. However, the peak in the control signal near the setpoint change after 650 seconds is likely caused by a coinciding migration. The control system is not aware of when or to where a migration will occur and does not attempt to mitigate its effects.

Active constraints

With the basic controller function and software migration established, it is time to look at a use case where there is potential for the controller to take advantage of the edge cloud. In the following experiment, there is a constraint set on the control signal to the plant. Albeit being a synthetic exercise, it is not an unreasonable action, since limits in control signal might be used to, for instance, reduce actuator wear and to avoid non-linear parts of the operating range. To make the associated optimization problem harder, the setpoint is changed to move the ball just short of the end of the beam. In combination with the constraints, this will cause a higher load on the MPC host node and small disturbances may cause the ball to fall off.

The experiment shown in Figure 7.10 applies this configuration. The graphs show the times series of the inputs to the controller, the processing time of the MPC, and the total latency from input to output of the measured nodes. The blue circles mark occasions when the MPC fails to find a feasible solution.

As constraints are tightened, it becomes increasingly hard to find a control signal sequence that moves the system from the present state to the target state, while staying within the bounds. This manifests as longer execution times for the optimization. As seen in the processing times of Figure 7.10, when the system has settled around a setpoint, the optimization is easy to solve and computationally light-weight. In these situations, the controller on the plant performs well. Latency and jitter of the networked controllers may cause them to deviate more from the setpoint.

Eventually however, the computational limitations of the plant causes

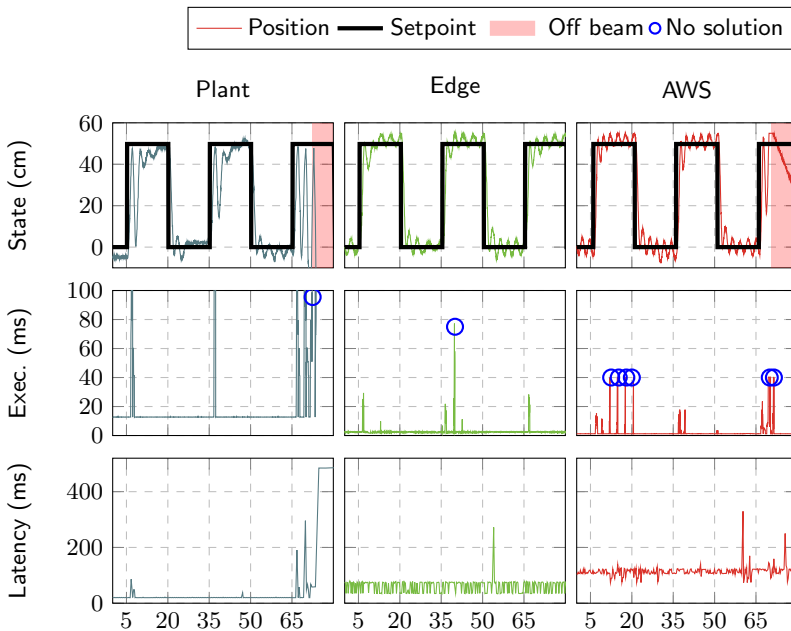


Figure 7.10 Time-series of experiments run with tightened constraints on the plant, RBS, and AWS nodes.

the ball to fall off the beam as a setpoint change occurs. Notice that the plant is not unable to move the ball to the position at the end of the beam, but eventually, model errors and noise become too large for it to handle. In contrast, the RBS node is able to operate without failure. Also note that on the AWS instance, despite its computational capacity, the controller fails to cope with the resulting latency and system jitter, and the ball falls off.

The processing time at the plant (on the Raspberry Pi), when the MPC fails to find a feasible solution, is close to an order of magnitude larger than that of the sampling time, represented on the second row by a line, which extends well beyond the top of the graph. This is representative of the computational problems experienced at the plant due to noise during a setpoint change. In contrast, an equal number of iterations consumes 80 ms on the RBS node and only 40 ms on the AWS.

With the position closer to the end of the beam and with reduced range in the control output signal, the controller repeatedly experiences non-trivial situations, which require additional iterations of the optimisation. At times, noisy readings make a tough situation even worse, and there may seemingly be no solution that keeps the ball on the beam. If a new evaluation can be

made quickly enough, the state may be such that the ball can be saved. On the AWS node, the communication delays increase the number of occurrences of these tough situations, and there are repeated problems of finding a solution. The speed of the AWS node, however, allows it to cope with many of these situations, since the combined processing and communication delay is much less than the compute time at the plant. Only the RBS node is in a position where it is able to handle the full range of the system noise.

7.6 Conclusions

This chapter presented an edge cloud research testbed for an IoT and heterogeneous cloud environment. The testbed includes 5G access technology, infrastructure, and software platform. A control application was deployed on the testbed controlling a time sensitive and mission critical process. The viability, performance, and system characteristics were evaluated. The evaluation shows that our reference controller can execute in a cloud native framework and viably be deployed on the edge cloud.

Works such as [Tärneberg, 2019; Pelle et al., 2019] show that cloud platform delays can be significant when combining services. In many cloud and IoT applications, it is therefore not unusual to consider delays in terms of hundreds of milliseconds. Calvin was created for IoT applications and the data-flow programming model of Calvin is reminiscent of how such applications are implemented. In relation to this, the platform in Chapter 7 is efficient. Nonetheless, the software platform is a major source of latency, and a lack of real-time support results also in large variation in the delay. The experiments operate at a sampling rate of 20 Hz, but there is potential for this to be pushed further with improvements in the software platform and control strategy. The experiments showed that the controller can benefit from the edge cloud and that the system, and the placement of the controller can be dynamically reconfigured during run-time without strictly sacrificing stability.

In addition to what can be classified as ordinary software improvements, such as reducing the overhead of message passing, there are also other features that could support controllers. For instance, support for replicating controllers and migrate active clones to other runtimes, to create redundancy. The result would resemble the cloud control system proposed in [Xia, 2012]. This can help reduce problems with delay spikes, but is costly in terms of resources. The result would also remain an instance of a direct NCS (Figure 3.1) and as such does not provide recovery if the network or application fails. The next chapter introduces offloading and the hierarchical NCS, while continuing to evaluate contemporary clouds in more detail.

8

Function Service Performance

Chapter 7 verified the soundness of the selected plant and controller in a context of low latency access networks and edge clouds. This established that the required bandwidth is reasonable and that an ordinary MPC can achieve a basic level of performance. We now turn attention to the data centers, function services, and the optimizing controller, progressively benchmarking the layers of ICT that constitute a cloud, from two infrastructure providers. This chapter adds the following contributions to the study of using cloud-native technology for feedback control systems:

- 1) It provides a study of the progression of latency in clouds from theoretical lower bounds to cloud-native application layers.
- 2) It benchmarks response times when offloading to a private cloud, a public cloud, and a massively parallel cloud service.
- 3) It proposes the cloud-augmented controller.
- 4) It replaces the PaaS application in Chapter 7 with client offloading, enabling the use of FaaS to implement the MPC in the cloud.
- 5) It provides identification of how prominent the detrimental effects are, when offloading feedback control using only arbitration.

The control problem is executed in two forms. The first form relies primarily on the remote controller and abruptly switches to a client controller when offloading fails. The second controller has a built in client mode, to reduce the use of the cloud and provide a graceful degradation. Extensions of these two forms are studied in Chapters 10 and 11.

This chapter is based on [Skarin et al., 2020c]

8.1 Objective

The goal is to study the response from a cloud native platform when executing a feedback controller using the client architecture in Figure 3.4. Chapter 7 showed that LuMaMi, a prototype 5G base station, can provide a reliable low latency air interface, and that the software platform was the dominant source of delay. With this knowledge, the software framework is simplified. Communication is run directly over the wired interface, as focus is shifted to the application and the cloud.

It is of interest to investigate the achievable response rate from the cloud with this new configuration. It is also of interest to observe the performance penalty from choosing technologies far up the software stack, such as representational state transfer (REST) interfaces, as the view that the cloud is unreliable and plagued by long delays often comes from judgments based on complex applications and/or obsolete technology. The gain from using general purpose technology is a cost effective average performance, and it is useful that applications can accept tail latencies [Bernat et al., 2002]. One question is how frequent and problematic outliers and self-imposed delays (due to processing time, large payload, many requests etc) are, for an offloaded control application. Therefore, we want to investigate if common assessments of cloud latency are overly pessimistic.

From Chapter 7 and [Pelle et al., 2019] we find that control code execution time is expected to be significant in a cloud application. It is therefore also of interest to investigate the effect of this load on the processing time properties. Complexity is reduced as much as possible to draw conclusions specifically for the MPC and find the achievable lower bounds on response times. This chapter limits the self-imposed delay to the optimization processing time, since there is no redundancy or other parallel activity in the controller, and no payload variation¹.

To make it easier to refer and reason about the objectives, three assumptions are investigated:

ASSUMPTION 8.1

Cloud-native platforms are composed of many systems-of-systems. This results in a delay with greater variance for each successive system or OSI layer. This is referred to as *platform noise*. □

ASSUMPTION 8.2

Control applications of the type in Figure 3.4, with a necessary frequency of at least 15 Hz, can benefit from offloading to the cloud using ordinary cloud-native services. □

¹The response to a large number of requests is part of the measurements in Chapter 10, where the controller implements multiple parallel requests and it is necessary to find a delay distribution for this scenario.

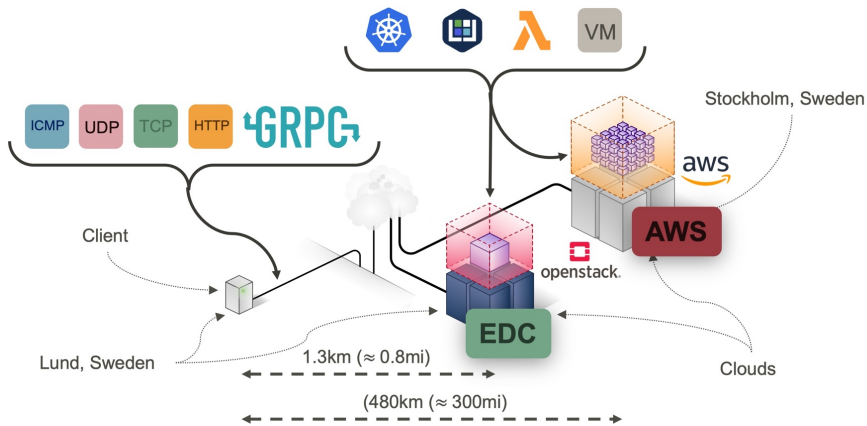


Figure 8.1 Cloud benchmarking infrastructure. The client is located on the Swedish University Network Lund (Lunet). EDC is the Ericsson Research Data Center (also referred to as ERDC).

ASSUMPTION 8.3

The targeted cloud control system incorporates multiple dynamic systems, and cannot be trivially predicted. In extension, combining results from studies of individual components is at high risk of leading to incorrect assessments of the suitability of the cloud. □

8.2 Platform

The platform that was used for these experiments is shown in Figure 8.1. The distant, hyperscale cloud in Frankfurt, Germany, from Chapter 7 is replaced by a similar cloud in Stockholm, Sweden. The plant's geographical location is on the same campus as in Chapter 7 but it is now wired directly to the Swedish University Network (SUNET) in Lund. ICMP, UDP, GRPC etc are standard communication protocols. The icons linked to the two clouds represent different deployments for the offloaded optimization.

The two locations represent a public cloud with an expected higher RTT and a very large and highly tuned infrastructure, and an edge cloud with an expected minimal RTT, comprised of a much smaller DC. The two are included for two reasons. One is to observe the trade-off between RTT and compute capacity, if any. The second is to make it possible to distinguish the impact of the cloud from other system components, such as the client implementation and intermediate networks. In summary:

ERDC is an OpenStack-based DC in Lund, Sweden hosted as a research platform by Ericsson (1.3km (approx 0.8mi) from the plant). It is labeled

EDC in Figure 8.1 and is representative of a proximal private cloud, alternatively, an Edge DC.

AWS eu-north-1 (a reasonably proximal AWS region) in Stockholm, Sweden (480km ($\approx 300mi$) from the plant) which is a representative of a capable COTS public cloud DC.

The plant is connected to the cloud infrastructures with a high-bandwidth and low-latency national back-bone network. The RTT is measured for a set of communication technologies; Internet Control Message Protocol (ICMP) echo, User Datagram Protocol (UDP), Transmission Control Protocol (TCP), gRPC (a cross-platform RPC framework), REST, and FaaS. They are representative of a journey though the Open Systems Interconnection (OSI) stack. At the application layer, Hypertext Transfer Protocol (HTTP) 1.1 and 2 are used in the form of REST and gRPC.

A minimal *echo response* is implemented for each communication technology and hosted on VMs (EC2 t3.micro and OpenStack c1m05) and containers (Hosted on equally dimensioned Kubernetes (K8S) clusters), when technically permissible. VM and container deployments will from now on be collectively referred to as IaaS. The combination of communication technology, service abstraction, and run-time paradigm as a *deployment*.

The MPC controller is implemented in Python using CVXGEN [Mattingley and Boyd, 2012] and adapted to the above set of deployments, so that its response time can be compared to the RTT measurements. The control client interfaces with the cloud using REST. This makes it directly compatible with the FaaS deployments. Towards Lamda, the function is requested using REST and native AWS Boto3 calls. When deployed directly on a VM, Python Flask is used in the workers to handle incoming requests.

For reiteration and for conducting the experiments at scale, a custom, automated deployment platform was constructed. With that platform, it is possible to reproduce the experiments and consistently observe the behavior of the system over long periods of time.

8.3 Experiments

A set of experiments were designed to test Assumptions 8.1 to 8.3. They are as follows:

Baseline RTT Investigates the minimal time it takes to reach the cloud, and go back again, at each successive layer in an infrastructure stack. These experiments are fundamental to testing Assumptions 8.1 and 8.2. The RTT experiments quantify the latency cost of accessing the cloud and finds the latency profiles for a set of communication technologies and cloud infrastructures.

Computational complexity vs. response time Measures the RTT as well as the controller processing time. These experiments explicitly test Assumption 8.2 and determine if Assumption 8.1 is load dependent. To create a varied load, the state of the plant and the MPC controller's prediction horizon is varied. To recreate identical conditions for each deployment, three static workload intensities were determined through experimentation; light, medium, and high. They are detailed below.

Feedback control Using the findings from the above experiments, we proceed to test Assumption 8.3 and evaluate the experimental feedback control system presented in Section 8.6. The nominal controller is investigated using setpoint changes. The performance evaluation criteria is the ability of the controller to stay within the constraints and to quickly reach the setpoint.

The experiments run in batches, with each batch executed as quickly as possible on a single thread, on an isolated computer at the campus. Although the cloud can be used for massively parallel computations, it is important to run single, serialized requests to find the shortest response times. There are ten requests in each batch. Because FaaS is used in the experiments, one request is sent initially and disregarded to remove *cold starts* from the data. In FaaS, a cold start happens when a request arrives for a function that is not yet loaded into memory. If there is a long idle period between requests, the function may have been taken out of memory and has to be loaded again. Due to the interest in lower bound performance and continuous sequences of offloaded control, cold starts are cleaned from the data. In case of the experiments in Section 8.5, these are executed a few times every minute over several days.

Performance metrics

The cloud performance is asserted in terms of a request's RTT and processing time, measured in milliseconds. Medians and percentiles are used to show and compare latency distributions. For the feedback control experiment, three metrics allows an assesment of the control performance over time. They are all relative to the stand alone performance of the client LQR (Figure 3.5), which has been tuned to stay within the physical constraints of the plant. The ball and beam plant is used, and focus is on the critically controlled variable x_1 , i.e., the ball's position. The constraints are slightly tightened in the MPC to provide a bit of margin. Low values are desirable for all metrics.

Relative Accumulated Violations (RAV) A measure of the total violation of constraints. c_0, c_1 are the upper and lower constraints, respec-

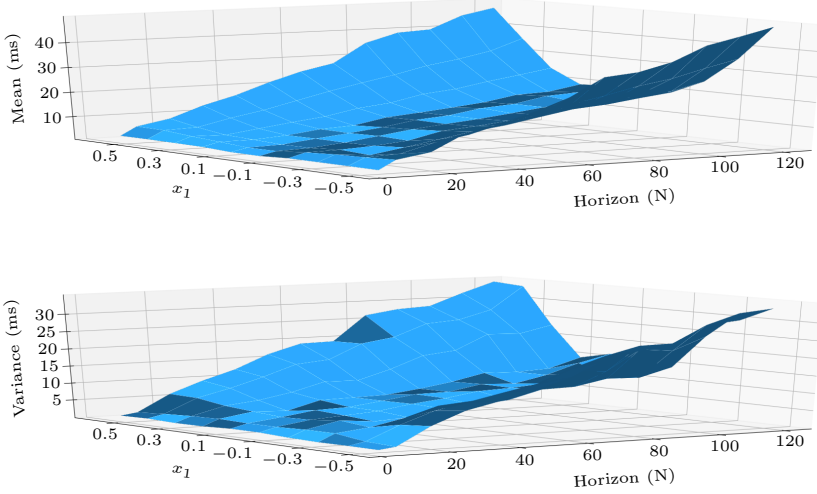


Figure 8.2 Processing time mean and variance for various horizons and state errors.

tively.

$$\frac{1}{T} \sum_{k=0}^T \sum_{j=0}^1 \max(0, |x_1(k)| - |c_j|) \quad (8.1)$$

Relative Accumulated Error (RAE) A measure of the controllers ability to reach the requested setpoint. r is the setpoint.

$$\frac{1}{T} \sum_{k=0}^T |x_1(k) - r(k)| \quad (8.2)$$

Relative Maximum Constraint Violation (RMCV) Measures the maximum distance from the constraint of the system state over time. \vec{x}_1 is an array of all state samples. c_0, c_1 are again, the upper and lower constraints. Subtraction and taking the absolute value, is done element wise.

$$\max(|\vec{x}_1| - |c_0| \cup |\vec{x}_1| - |c_1|) \quad (8.3)$$

8.4 Computational load

A benefit of the MPC is that its computational requirements can be scaled. The performance and load of the controller depends on the design parameter

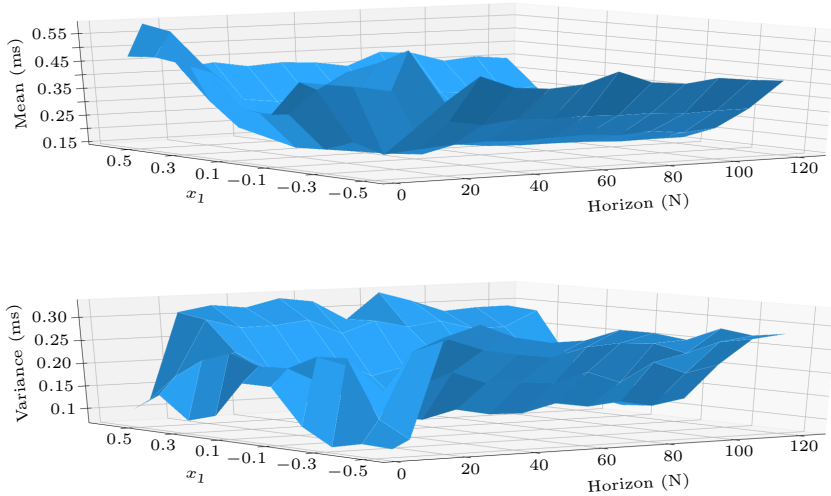


Figure 8.3 Processing time mean and variance for various horizons and state errors scaled by the horizon.

N in Equation (4.26). The solver that computes Equation (4.28) (or solves the general Equation (4.1)) requires an unknown and variable number of iterations to find the optimal control sequence for a given N , depending on the state of the plant. This is illustrated in Figure 8.2 and Figure 8.3, which show measurements of processing time, in different plant states and with different horizons using the QPgen optimizer. The plots in these figures are created from averaging a large amount of requests, several in each state, towards one of the cloud services (HAProxy with a PyQPgen optimizer). The vertical y-axis shows mean values in the upper plots and variance in the lower plots. The x-axis shows the error in x_1 and the z-axis the horizon. For each combination of error and horizon, multiple requests have been processed over a few combinations of x_2 and x_3 .

The curvature in the figures shows variation in load when varying position error and the horizon. A mostly linear increase in processing time and variance in response to increased horizon is visible in the first figure. The second figure verifies this observation by taking the computation time τ^c and dividing it by N . This will be compared with measurements on the different platforms.

In the experiments, a range of horizons are observed and three load scenarios are defined based on the number of required iterations. The load scenarios

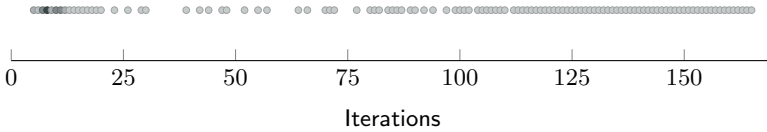


Figure 8.4 Registered number of iterations in the optimizer. The darker cluster at the lower end is where the bulk of requests are.

represent the variation in Figure 8.3. Running the controller for some time reveals a range of iterations that the solver uses with $N = 30$. This result is shown in Figure 8.4. From this, three plant states are determined and used to define the load scenarios. The three load scenarios are

- 1) *light* with 5 iterations,
- 2) *medium* with 38-82 iterations, and
- 3) *heavy* with 93-152 iterations.

There is a range of iterations specified for the medium and heavy cases. The reasons for this is that as N changes, the number of iterations can also change, even through the plant's state is fixed. Thus, in the experiments, the number of iterations will not be equal for all cases of N , and the increased load might not be linear. However, the number of iterations primarily change from $N=5$ to $N=10$. After $N = 10$ the increase is small with each step in N , and does not have a significant impact. It is worth keeping in mind, that an increased horizon can increase the number of iterations, especially when N is low.

8.5 Cloud stack latency progression

We proceed to evaluate Assumptions 8.1-8.3 through results from the experiments detailed in Section 8.3. The used deployments, endpoints, and communication protocols are listed in Tables 8.1 and 8.2. Descriptions in Table 8.1 relate components to the overview Figure 2.3 on page 42.

Upper frequency bounds and noise floor

With the goal of using cloud-native platforms, testing Assumption 8.1 entails answering whether or not a significant amount of delay and variance are added for each layer in the cloud. Additionally, evaluating Assumption 8.2 requires an assessment of the frequency at which the cloud is able to reply timely. Figure 8.5 shows the measured RTT towards the various layers in the two cloud infrastructures. For completeness, at the bottom on the stack, the RTT for light in fiber is included, and the theoretical RTT of a single IP

Table 8.1 Deployments and endpoints used in the experiments and in Figure 8.5.

Front End (FE)	Entry point to the DC. Reached using the clouds global domain name and represented by the cloud provider interface (World) in Figure 2.3 on page 42.
Virtual Front (VF)	The virtual machine serving as entry point of a cluster. Represented by the Service VM (load balancer) of the green cluster in Figure 2.3 on page 42.
Worker Node (WN)	Requests to a virtual machine behind the Virtual Front. Represented by are the worker VMs of the green cluster in Figure 2.3 on page 42.
Kubernetes (K8S)	Request to a Kubernetes node. This is represented by the worker and master VMs of the purple cluster in Figure 2.3 on page 42.
HAProxy (HAP)	A request to a HAProxy cluster, represented by the green cluster in Figure 2.3 on page 42.
Kubeless	A request to a Kubeless function, a custom Function-as-a-Service executing on a Kubernetes cluster. Function A in Figure 2.3 on page 42.
Elastic Load Balancer (ELB)	Requests going through an external load balancer. This is provided as a cloud service and does not execute in a managed VM.
Lambda	Large scale, Function-as-a-Service

Table 8.2 The communication protocols in Figure 8.5.

IP frame	The theoretical value of transferring a minimal Infrastructure Providers (IP) packet, the lowest level software protocol.
ICMP	A minimal request using the Internet Control Message Protocol. Represents the lowest level of the IP stack. Not used to transfer data. This is a <i>ping</i> request.
UDP	A connectionless UDP request with no guarantee of delivery and ordering. Suitable for real-time applications that prefer dropped packets over retransmission delays.
TCP	Uses the connection-oriented TCP. Provides reliable and ordered requests.
gRPC	A modern Remote Procedure Call (RPC) framework endorsed by the Cloud Native Computing Foundation (see Section 2.2). Uses HTTP/2 to transport data.
REST	Uses REST over HTTP. This is a very common way of interfacing with cloud services.

packet. This provides a RTT progression from the theoretical latency of light in fiber to highly abstract FaaS deployments.

To discuss the results, we assume the requirement that the latency is lower than one sampling period. With this assumption, all deployments are compatible with Assumption 8.2, i.e., a response time of less than approximately 66 ms. There is room for computations even at the 99th percentile of the worst case, Lambda, but a large portion of the response time would be consumed in flight for the AWS REST deployments. The portion of time that must be available for computations is case dependent, but using Chapter 7 and Figure 7.7b as an example, the cloud finishes the optimization roughly five times faster than the device. If the device takes exactly the full 66 ms to complete, the cloud will finish in around 13 ms leaving more than 50 ms to transfer the result. If the frequency on the device is 50 Hz, the response time has to be 16 ms. Doubling the frequency from 15 Hz to 30 Hz but assuming the same execution time, the cloud has to respond within approximately $33 - 66/5 \approx 20$ ms. With this reasoning, Lambda leaves little room for frequencies above the requirement of Assumption 8.2, while the data suggests there is room for more than doubling the frequency across all deployments in the proximal DC. We see from the percentiles that the platform noise increases in absolute terms when moving up the stack, as assessed by Assumption 8.1. In the lower layers (ICMP, UDP, TCP, gRPC), it looks similar for the two infrastructures, and the noise is large in relative terms for the municipal DC. In absolute terms, in relation to Assumption 8.2, the noise looks sufficiently small.

Based on the observed RTT, the achievable response frequency from the cloud varies between 100 Hz to 500 Hz and 33 Hz to 80 Hz, in ERDC and AWS eu-north-1, respectively. The theoretical upper bound is the RTT in fibre, which allows for 100 kHz and 156 Hz, in ERDC and AWS eu-north-1, respectively. However, UDP is the first point of access to the compute resources in the cloud, and frequencies of 500 Hz and 80 Hz are achieved for the two cloud infrastructures. The discrepancy from the theoretical IP frame latency in fibre to ICMP (first entry into the stack) is an indication of the overhead in the intermediate networks. The increase is roughly 2-fold for AWS eu-north-1 (from six to twelve milliseconds). For ERDC, where the theoretical values are in the order of microseconds (this appears as zero in the figure), the increase is 90-fold. Continuing to look at ICMP, at an almost 370 times greater distance from ERDC to AWS eu-north-1, the latency is a factor 8 higher towards the distant DC. The RTT over UDP is a factor 4 greater in AWS eu-north-1 compared to ERDC. Note also that the RTT's variance in the case of ERDC is significantly larger.

When communicating with the compute resources inside a DC, the most significant degradation, in either DC, is when stepping into the application layer, such as adding an HTTP 1.1 header. REST, a common service in-

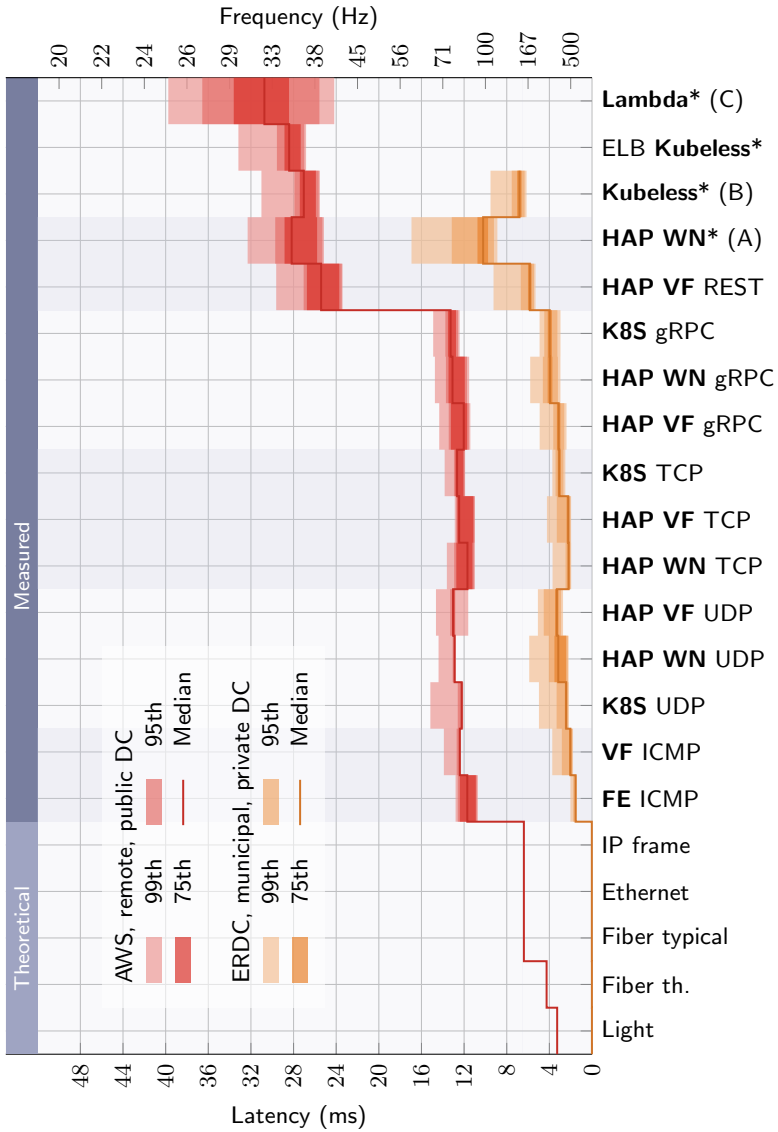


Figure 8.5 Echo request progression from theoretical round-trip delays to cloud service response times. Entries marked with * are accessed using REST. See deployments and protocols in Tables 8.1 and 8.2.

interface, has twice the RTT compared to the TCP, the underlying transport protocol employed by HTTP. Further, gRPC takes advantage of a newer HTTP standard, HTTP/2, and therefore achieves a lower response time, almost half. REST APIs implemented with HTTP is a ubiquitous technology among cloud applications and is the common approach for accessing FaaS platforms. As a design criteria, FaaS is the targeted service model and as outlined above, we can continue to investigate FaaS as an alternative compatible with Assumption 8.2. It is seen from Figure 8.5, that the layers of software platforms and opaque infrastructure management policies can increase the latency in comparison to ICMP, 5 and 2.5 fold, in ERDC and AWS eu-north-1, respectively.

There is no reason to disqualify any deployments based on these measurements. What should be noted though is the large increase in latency towards the distant cloud when moving to the REST interface. There is also a very noticeable increase when moving from the Virtual Front End (VF) to the Worker Node (WN) when using REST. It is also clear that Lambda, the only large scale function service, exhibits further latency and large variation. On the other hand, there is no notable difference between the packet based, unreliable UDP and the connection oriented, reliable TCP in these tests. If anything, TCP has the upper hand. This is by no means a benchmark between these two protocols, but it is clear that our concern should be with moving up the stack to the REST interface, where we have to use TCP.

There are some observation to be made concerning the two clouds. We saw in Chapter 7 that the performance of the VMs was better on the instances in AWS. We can see something similar again looking at the larger increase in the mean for ERDC when moving from HAP VF to HAP WN, and the clearly larger relative variations. When forwarding to HAP WN, the request data has to be processed by two virtual machines (VF and WN). Another difference is the increased latency from gRPC to HAP VF REST. It is certainly much larger in absolute terms towards AWS, but also significantly in relative terms, compared to ERDC (roughly 1.5 vs two-fold in the mean, and a large increase in the 75th percentile). On the positive side, moving to REST towards the municipal DC does not significantly increase latency. We can assume that this discrepancy is due to intermediate networks because of the distance to AWS, but we do not know from the data how much of the added delay that happens inside the cloud.

Keep in mind that these numbers are lower bound since the request payloads are minimal and no valuable computation is done in the cloud. Next, the systems are subjected to load in order to evaluate Assumption 8.1 and the expected response times for the experimental control system.

Noise floor and the response to load

Having established the upper bound response frequency and minimum variance in Section 8.5, we are ready to evaluate the response frequency and platform noise when executing the controller. Since the REST and FaaS deployments are our targets, we proceed with that subset of deployments, namely,

- (A) REST over High Availability-proxy (HAProxy)
- (B) Kubeless
- (C) AWS Lambda on eu-north-1.

These deployments are marked as (A),(B) and (C) in Figure 8.5. There are five deployments in total, (A) and (B) have deployments in both data centers, while (C) is a service only available on AWS. The interests is in determining the response rate that can be achieved in these deployments, and how the load affects the platform noise.

The plots in Figure 8.6 show response times per controller horizon for each load scenario and deployment. A linear increase in the mean computation time is expected with an increasing horizon (N , on the x-axis). This is clearly visible in Figure 8.6, and apart from Lambda, there are only small discrepancies from this linear response.

There are notable differences in response times between the deployments. Although the processing times visually start off at a similar level, the gradient on ERDC, going from light to heavy load, is significantly steeper than on AWS, on average 2.3 times greater, consistently across the deployments. The ERDC variance increases with the load. This is particularly evident in the ERDC Kubeless deployment. However, on AWS the trend is not as pronounced. Instead, there is an initial increase followed by a decline in variance for AWS HAProxy, which peaks between $N = 15$ and $N = 35$ for both processing and response time in the heavy experiment. It is also clearly visible that the Kubeless deployment on ERDC incurs a large response time penalty and that processing times are affected negatively, especially in terms of variance. This is reminiscent of the aforementioned platform noise.

There is a large offset in response time between ERDC HAProxy and ERDC Kubeless. This is in contrast to what is observed in Figure 8.5, where Kubeless has a faster response. The same effect is seen for AWS although less pronounced. Looking at the light load and small N , the HAProxy deployment is inline with what is expected, given Figure 8.5. The Kubeless deployments for some reason seem to incur a penalty. This may be related to a configuration issue but can also be attributed to a form of platform noise, especially since the Kubeless deployment is the more complex of the two.

The visually most striking result is observed for Lambda. In the light load scenario, the processing time is comparable across all deployments, but

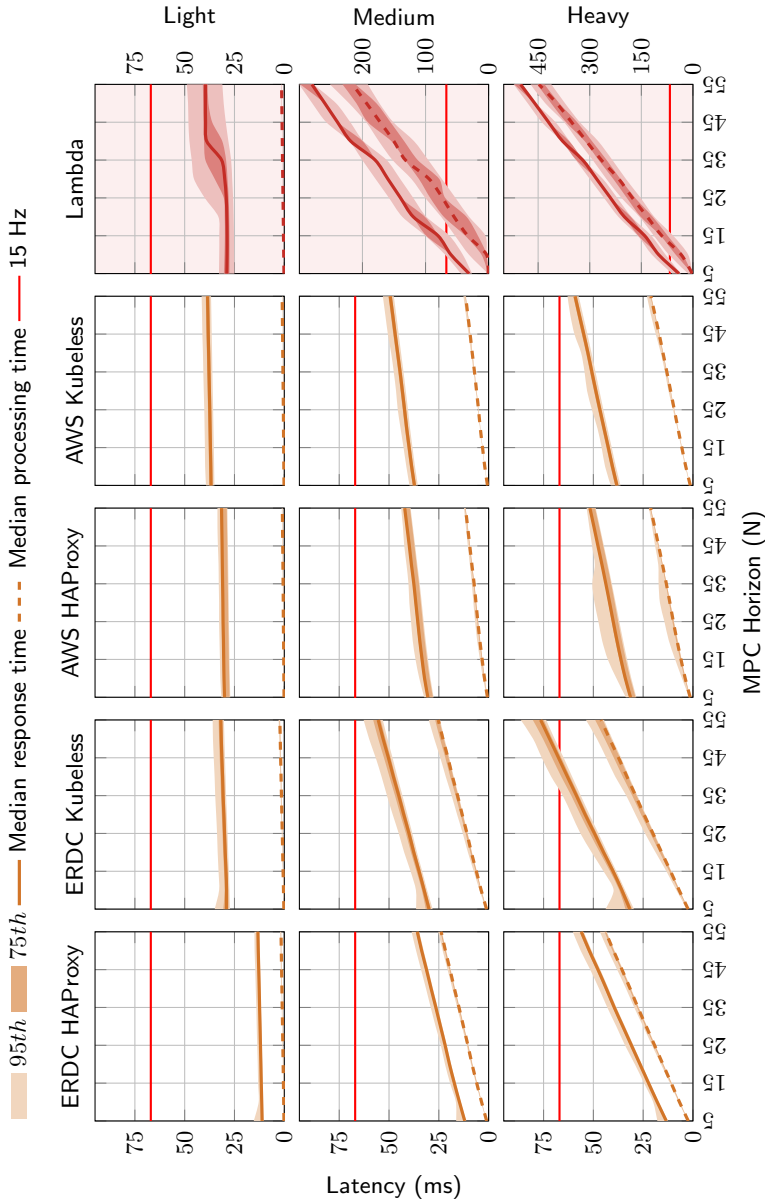


Figure 8.6 Latency vs. horizon. The red line is at specified minimum 15 Hz (67 ms). Percentiles around response time (solid) and processing time (dashed).

the Lambda response time has a clear mode shift between $N = 30$ and $N = 40$. Note also that the variance in response time is significantly larger than for the other deployments. The medium complexity case presents the same attributes. Here, multiple modes are present, at around $N = 15$ and $N = 35$. Also, the processing time is 20 times that of HAProxy and Kubeless, and Lambda has a significant variance, across the range of horizons. These phenomena point to a complex cloud-native platform, built to be shared by many at a large scale, handle massive parallel requests, and not carry a single real-time thread.

Except for Lambda and the response time difference between HAProxy and FaaS, the results in Figure 8.6 mostly points to a performance difference between the allocated VMs in the two different clouds. To get further insights into the first four deployments, they were subjected to a static load for a few hours. Figure 8.7 shows the median of the processing time, using a sliding window, for a load scenario with $N = 30$ and more than 300 iterations in a single optimization. The median is accompanied by area plots that show the minimum and maximum processing time within the window. What is clearly visible here is that there is a very notable difference in processing time variation between the HAProxy and Kubeless deployments. There also seems to be more platform noise in the ERDC HA deployment than in the AWS HAProxy deployment. This could be a direct consequence of the much longer (double) average processing time. It is also notable that although the median is almost a straight line for AWS HAProxy, the brown area plot in the background shows that there are occasions of extended processing time. Some variation and sporadic delays can be expected from the non-real-time property of the system, and equate to any ordinary general purpose computer. However, the bump in processing time in AWS HAProxy at 5.5 hours is different. Here, the median changes over a longer period of time which indicates that temporarily (although for several minutes) the available computation time decreases.

Another interesting feature of Figure 8.7 is the difference between the HA Proxy and the Kubeless deployments. Since these deployments are all on different VMs, it is possible that differences in hardware and over-provisioning (i.e. sharing with other users) give rise to such discrepancies. However, note that the effect is consistent and equally present on both infrastructures. This enforces Assumption 8.1. Even though the experiments execute one service at a time, the additional installed routing, software, monitoring etc that goes into a Kubeless deployment incurs a distinguishable overhead.

To sum up, there is a performance gap between Lambda and the other deployments. In a large scale public service such as Lambda, variations are to be expected over time, due to for example congestion, but the results also show service dynamics in the short term, based on the request rate and load. Due to the rapidly diminishing usefulness as the load increases,

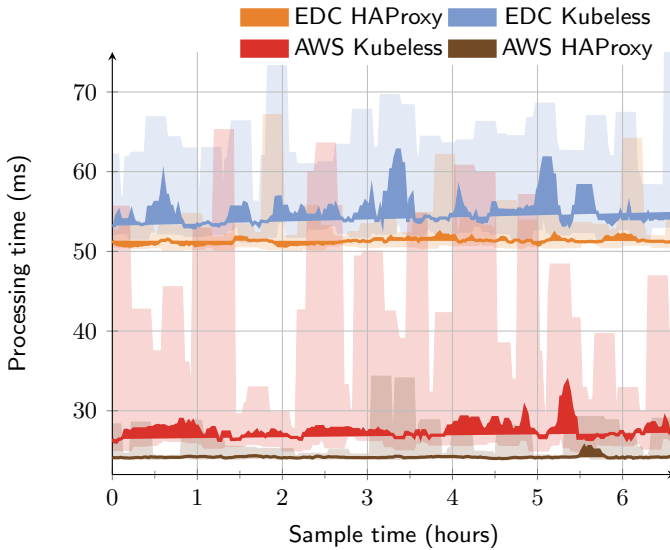


Figure 8.7 Processing time of a static load evaluated with two requests every 3 minutes, separated by a 0.5 seconds delay. Solid lines show a sliding window median over 10 samples. Shaded areas show the max and min values in the window.

Assumption 8.2 does not seem to hold for Lambda. If control frequencies are selected close to the median, Assumption 8.1 should be relevant for the end result, in which case our cloud native FaaS is at a disadvantage compared to HAProxy, as seen from the variance observed in Figure 8.7.

In terms of the example control application, when requiring only a few iterations (light load), the controller can be actuated at a rate of between 25 Hz to 90 Hz depending on the deployment, for all horizons. For the medium and high load scenarios the response rates are between 4 Hz to 90 Hz and 2 Hz to 66 Hz, respectively. This assessment does not take into account the probability of consecutive long delays or the effect that the variance causes.

8.6 Feedback control

Plant simulation

The MPC controller is derived from the same definition as in Chapter 7. It implements hard constraints and is nominally stable. In addition to the risk of not providing timely control signals, this controller can also become infeasible due to model errors, measurement errors, and external disturbances. Since we are not examining the robustness of the controller, the plant is simulated.

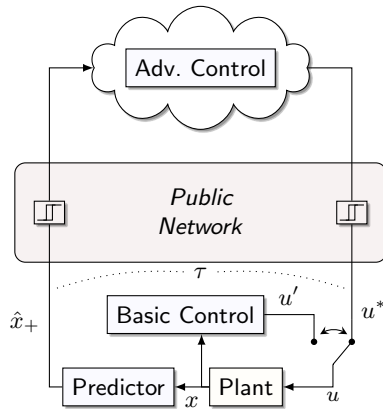


Figure 8.8 Schematic view of primitive critical control over the cloud. The switch is open for advanced control as long as the cloud is responsive. The basic control is abruptly switched in when there is no input from the cloud. The second method in the chapter replaces the switch with client side logic.

This also removes the technical difficulties of achieving real-time on the client in the experiments. The simulation of the plant uses an ordinary first order approximation, i.e., the forward Euler method,

$$\begin{aligned}
 z(j+1) &= z(j) + \left[\begin{array}{c} v \\ -\sin(\theta) \cdot g \cdot m \\ u \end{array} \right] / \xi \\
 z(0) &= x(t) \\
 x(t+h) &= z(s)
 \end{aligned}
 \tag{8.4}$$

where $x(t)$ is the plant state and ξ is the number of samples, i.e. the precision of the simulation. Small errors between this simulation and the discretized controller are enough to create problems around the constraints.

Control

A refined, primitive version of the controller from Figure 3.4 is shown in Figure 8.8. The client has been replaced by a basic controller and a predictor. An advanced controller is implemented in the cloud and switching logic select between the local control signal u' and the result from the cloud u^* . The two controllers independently try to control the plant and directly manipulate its input. The switch in the figure illustrates that there is arbitration between the controllers.

```

1:  $k \leftarrow 0$ 
2:  $u \leftarrow 0$ 
3: for  $\infty$  do
4:    $x \leftarrow \text{Read}$ 
5:   if feasible solution and  $t' + \tau < kT_s$  then
6:      $u \leftarrow \text{MPC}(\hat{x}_+)$ 
7:   else
8:      $u \leftarrow Kx$ 
9:   end if
10:  Apply  $u$ 
11:   $\hat{x}_+ \leftarrow f(x, u)$ 
12:  Send  $\hat{x}_+$  to cloud
13:   $t' \leftarrow t$ 
14:   $k \leftarrow k + 1$ 
15:  while  $t < kT_s$  do
16:    Wait
17:  end while
18: end for

```

Algorithm 8.1: Primitive client arbitration

The experiments use two control applications that differs with respect to how the arbitration between the basic controller and the advanced controller is done. In the first, the basic controller serves as an ancillary controller, taking action when the advanced controller fails to respond, i.e., when u^* does not arrive from the network. This client implements Algorithm 8.1. At Line 11, a prediction is made using the model of the plant, i.e. at time k the networked controller uses a predicted state \hat{x}_+ to calculate a control signal $u^*(0)$ for time $k + 1$. This prediction provides a time frame in which the controller can send requests, allow optimizations to finish, and receive responses from the cloud Adding this one step delay creates a static delay for the controller, which is easy to account for. A one step delay is also a natural way to handle the variable execution time delay caused by the MPC controller itself [Findeisen and Allgöwer, 2004], as for example proposed in [Cortes et al., 2012; Chen et al., 2000].

In the first of the two modes, the controller always requests support from the network. At every iteration of the controller, a request is sent to the network, and if a response arrives in time before the next iteration, the result from the MPC is applied. When the request latency is too large, it uses the ancillary control. In the second mode, the controller does not always use the MPC, and when it does, it transitions differently to the local control mode in case of failure. This is shown in the more involved Algorithm 8.2. In this mode, the client does not execute the MPC when the plant state is inside the terminal set of the MPC. To ensure reliable control, the terminal set

```

1:  $k \leftarrow 0, i \leftarrow 0, u \leftarrow 0, u^* \leftarrow (0, \dots) \in \mathbb{R}^1 \times N, x^* \leftarrow (0, \dots) \in \mathbb{R}^n \times N$ 
2: for  $\infty$  do
3:    $x \leftarrow \text{Read}$ 
4:   if  $u = \text{NaN}$  then
5:      $u = Kx$ 
6:   end if
7:   Apply  $u$ 
8:    $u \leftarrow \text{NaN}$ 
9:   if  $x \notin \mathcal{X}_f$  then
10:     $\hat{x}_+ \leftarrow f(x, u)$ 
11:    Send  $\hat{x}_+$  to cloud
12:     $t' \leftarrow t, k \leftarrow k + 1, i \leftarrow i + 1$ 
13:    if  $i < N$  then
14:       $u \leftarrow \kappa_2(u^*, x^*, i, \hat{x}_+)$ 
15:    end if
16:    while  $t < kT_s$  do
17:      if feasible solution and  $t' + \tau < kT_s$  then
18:         $x^*, u^* \leftarrow \text{MPC}(\hat{x}_+)$ 
19:         $i \leftarrow 0$ 
20:         $u \leftarrow u^*(0)$ 
21:      end if
22:    end while
23:  end if
24: end for

```

Algorithm 8.2: Dual mode with graceful client switching

should be an invariant set for the basic controller, as detailed in Section 4.4. This mode also receives and stores the predicted states and control signals of the MPC (Line 18). This is used by the function κ_2 at Line 14. The details of κ_2 are discussed in Chapter 10. For now, the important part is that this function uses the previous prediction from the MPC and combines them with the response from the basic controller to implement the switch.

Algorithm 8.1 examines how well a naive approach works. This will tell us something about the detrimental effects of the deployments to control of the reference plant, and studies Assumption 8.3. The second mode contrast these results with an approach that aims to provide gracefully degradation (Algorithm 8.2). The other difference between the modes is that the second controller only executes the MPC on setpoint changes (or, in practice, due to a large disturbance) and returns to local client control when the state has sufficiently settled.

Table 8.3 Performance measures for the feedback control system. All measures are in relation to the local LQ regulator. A * marks the second controller implementation, as described in Section 8.6.

System	Cloud	$E(\tau)$	RAE	RAV	RMCV
Baseline	0.00	0.00	1.00	1.00	1.00
Lambda	0.16	60.82	0.88	0.68	2.47
ERDC HAProxy	0.95	15.66	0.60	0.01	2.28
ERDC Kubeless	0.82	34.67	0.59	0.01	1.90
AWS HAProxy	0.96	31.95	0.60	0.01	2.28
AWS Kubeless	0.85	39.78	0.59	0.01	2.28
AWS HAProxy*	0.40	15.40	0.90	0.09	0.31
Lambda*	0.40	14.40	0.90	0.09	0.50
No Delay	1.00	0.00	0.60	0.00	0.00

Examining the closed loop control

We proceed to evaluate the closed loop as specified in Section 8.3. Kubeless and AWS Lambda represent our cloud-native FaaS, the primary target of interest. The set also includes the IaaS deployment for REST, since this is a compatible and comparable service. The basic controller is implemented as an LQR that has been tuned to not violate the state and input constraints. This local controller acts as the performance baseline, since it can be used without connectivity to the cloud. The advanced controller implements the MPC in Section 4.4. All deployments, also Lambda, run at 20 Hz. This is the same frequency that was used in Chapter 7. The horizon for the first control mode is set to $N = 50$. In line with being the naive approach, this sets a large horizon in an attempt to achieve good performance. Based on the findings so far, it should allow successful operation on all IaaS services, while Lambda should be able to provide support for the lower loads. The second mode is aimed at being reliable and tuned for use with the cloud. Therefore, the horizon is reduced to $N = 20$ and responses from AWS HAProxy and Lambda are examined. According to Figure 8.6, the second configuration should allow an almost perfect response from AWS HAProxy, while Lambda will remain problematic.

The experiment results should tell us at least three things:

- 1) whether the reference system works as expected in terms of meeting the expectations from the benchmarks,
- 2) the response of the system when only the lower loads are able to respond timely, which is what we expect to see from Lambda, and
- 3) how well the switching mechanism works.

Table 8.3 provides an overview of the control outcome. The table shows the baseline controller (the LQR) in the first row and the response without delays in the last row (i.e. an ordinary MPC with a one step prediction and no network). The starred rows execute the second controller mode. The second column shows the fraction of time the networked controller was used.

The expected response time for each deployment, $E(\tau)$, is presented in the table's 3rd column. From Table 8.3, most noteworthy is the AWS Lambda deployment. Here, only 16% of the requests get responses from the networked controller. This is in line with the results from Section 8.5, where the response times for AWS Lambda were often well above 50 ms, $E(\tau) = 60.82$. Further, the ERDC and AWS HA deployments utilized the cloud 95% and 96% of the time. These numbers are 82% and 85% for ERDC and AWS FaaS.

The RAE in Table 8.3, which shows the long term efficiency of the controller, is improved by all cloud-based networked controllers, as expected. In fact, the ERDC Kubeless deployment scores a 40 % improvement, same as the MPC without the network (bottom), at 82 % response rate. Even Lambda, with its low response rate, scores a 12 % improvement.

The networked controller knows the system's constraints and consequently attempts to strictly enforce them. There are two complicating disturbances that may cause this to fail: a small model error and network loss. The RAV metric measures the extent to which the controller fails to enforce the constraints by summing up the amount of constraint violation over the course of the experiment. A RAV value above zero is undesirable, but also hard to avoid. Small values are acceptable but being worse than the baseline (i.e. above one) would be unacceptable. AWS Lambda makes extensive use of the local controller and therefore gets a relatively high RAV. The four deployments on rows 3-6 score almost perfectly due to the high response rates from the networked controller.

Several deployments perform on par with ideal values (RAE of 0.6 and zero RAV). Observed with the same metric, Lambda also performs satisfactorily compared to the baseline. However, it is important that the controller achieves a low *maximum constraint violation*. This is measured by the RMCV metric, which tells us whether temporary degradation can cause system failure. As seen in the table, this is where the primitive cloud controller fails. A simulated RMCV above 2 indicates an unrecoverable error in the real plant, and out of the first six cases in Table 8.3, only the baseline and ERDC Kubeless manage to stay below this value. The ERDC Kubeless deployment does score an RMCV below 2 in the experiment but it is expected that it will fail eventually. In part, Assumption 8.3 states that switching cannot be assumed safe just because the two individual systems are reliable, and we see here how easy this is to verify. Even the limited loss of 4% for AWS HAProxy is enough to cause repeated failure during an experiment of 3.5 hours.

Before turning to the second mode, in the last two cases of the table, take

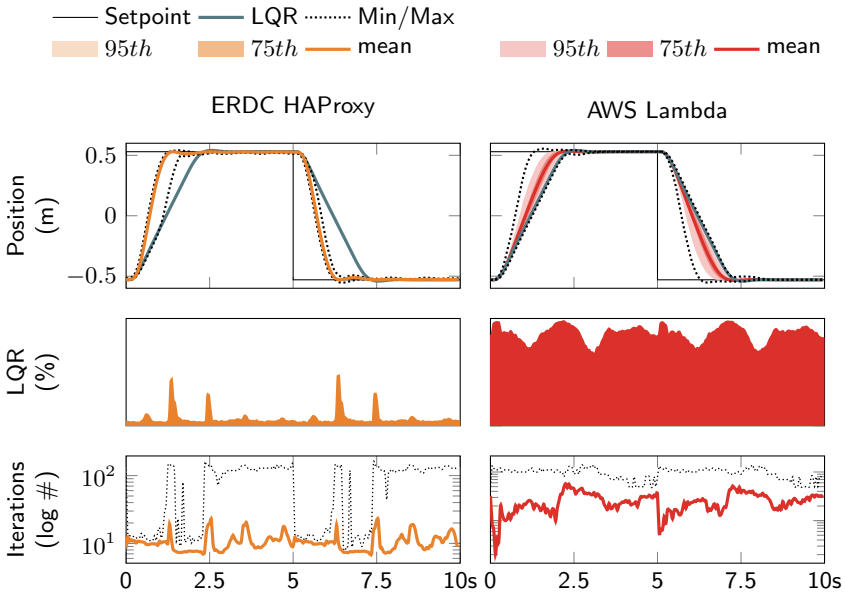


Figure 8.9 Statistics from responses in the HAProxy service of the municipal data center and the large scale public FaaS service. The setpoint changes in the upper plots are executed periodically over several hours which are then merge into statics for each sample in the periodic window. The second row shows the fraction of time the controller switched to the basic controller. The bottom row shows a log scale of the average and maximum number of iterations.

a look at Figure 8.9. In the experiments, step changes are executed repeatedly over a longer period of time. The figure shows the controller response over the periodic step change as statistics from all the step changes in the experiment, in terms of a mean response, a minimum and maximum response, and percentiles. The plot to the left shows that the ERDC HA Proxy deployment has a good mean response, close to its maximum. It also has very little variation but has, on some occasion, deviated from its mean, as seen from the dotted lines. The slow response on the rise, referred to as the minimum, is due to failed requests at the start of the step change. The constraint violation causing the RMCV value is observed in the dotted line (the minimum) in the step down to -0.5 . On the right we see that there is more variation and an overshoot, in the max response, on the part of Lambda. While its mean is better than the basic controller, this controller does not make very good use of all the requests sent to the cloud. The second row shows the fraction of samples that the basic controller was used. For ERDC HAProxy, there are

spikes in this figure at points when the position is close to and moving towards the constraint. This is not observed on the right, where request failure is common and the basic controller is used most of the time. The pattern repeats for the mean iterations of the optimizer, shown in the third row. The third row also show the maximum iterations as dotted lines. Here, we see on the left that extensive calculations can happen also in places where the basic control is not used frequently (2.5 s to 5 s and 7.5 s to 10 s) although the plant looks to have settled at the setpoint. In these time periods, the plant state will often be inside the terminal constraints, and it is not necessary to execute the MPC controller.

Now turn to the results of the second mode, in rows seven and eight of Table 8.3. These experiment were run for seven hours, twice as long as the previous experiments. Keep in mind the restricted use of the MPC, as mentioned above, and that this controller uses a shorter horizon to reduce its execution time. Ideal performance is not reached with this controller in terms of RAE, but in return it keeps RAV and RMCV low. For this reason, it achieves efficiency gains and remains reliable, for both deployments. The reason for the higher RAE is the shorter horizon. This controller will experience a slow response, as with the minimum response of ERDC HAProxy in Figure 8.9, because of the horizon. It cannot provide a feasible result until it finds one that has settled within one second. Increasing the horizon will provide a better RAE, at least on AWS HAProxy.

Looking at the differences between the results on Lambda, the efficiency (the RAE) of the two controllers is similar on this deployment. Because of the low load, the response time $E(\tau)$ for the second mode is similar to ERDC HAProxy on Lambda. We can assume from this that most requests do not fail because of delay. The number 0.4 in the Cloud column of Table 8.3 shows how often the cloud is used for control. Because there are few losses due to delay and infeasible optimizations, this number represents the request rate. Thus, the number reflects that the controller uses the basic controller around the setpoint, showing the reduced load compared to the first mode. In more detail, the first mode is sending requests 100 % of the time, but only gets useful responses 16% of the time. A mean response time of 60 ms is experienced. The second deployment receives useful responses to most requests, but it also only sends requests for roughly 40 % of its samples. It experiences a mean response time of roughly 15 ms. The first deployment achieves an improved error rate of 0.88, but also experiences critical constraint violations, and is loading the cloud with a lot of unused requests. The second deployment, with its lower load in computations and requests, achieves an improved error rate of 0.9, with no critical constraint violations, and is much more cost effective.

Because of its reliability and lower load, the latter of the two installments is preferred. However, its performance is limited by the horizon. This also means that the switching in Algorithm 8.2 was never or seldom used. Chap-

ter 10 explores this mode and expands it with flexible performance through a variable horizon.

8.7 Conclusions

This chapter has provided the following insight into using clouds and cloud native software for critical control of constrained dynamic systems.

- The first experiments looked at achievable performance of two clouds by measuring access delay. This provided a measure of expected latency and noise through the software stack of two clouds, from the theoretical delay in fiber up to the cloud native service.
- The measurements show the increased latency issues in higher layers of abstraction, but also how distance becomes increasingly relevant. The small increase in latency when moving to HAProxy REST in ERDC, and the respective significant change for AWS, shows how useful a distributed cloud infrastructure can be, especially far up the software stack.
- Combined with knowledge from the wireless interface, demonstrated in Chapter 7, we see that the access medium, i.e. a wired or wireless channel, is not the major issue. It is more relevant to consider classic software engineering practice of weighting the efficiency of low level designs towards the gains from higher levels of abstractions.
- The second set of experiments placed loads in the cloud by repeatedly executing pre-defined optimization problems. This showed a large gap between the true, large scale, FaaS service and personal IaaS/FaaS deployments. An additional experiment showed large variations in the custom FaaS deployment, compared to the classic HAProxy setup, and a notable service distribution in the otherwise steady AWS HAProxy deployment.
- The second set of experiments also showed modes in the delay caused by the FaaS service when the load was changed. The IaaS/FaaS configurations showed a linear mean response, in-line with the added load.
- The third set of experiments deployed the reference feedback control system, implemented to be compatible with FaaS. Results show that the basic latency assessment holds in practice, with distinctly measurable efficiency improvements, but it also identifies a challenge to provide techniques that eliminate or reduces the impact of an increased RMCV.
- Through an improved design the request load of the cloud controller was reduced while retaining measurable efficiency gains and providing

a reliable closed-loop system. The results show potential for using the cloud for temporary performance boosts, without real-time guarantees, in critical control loops, also at relatively high frequency.

With the insights gained from experiments with 5G infrastructure and cloud native implementations through PaaS and FaaS, the next part of the thesis explores ways in which to construct control applications. The focus is on using the cloud, and distributed clouds to improve performance, simplify development, and mitigate latency and uncertainty issues. The most practical way to study these concepts is to continue using FaaS and offloading as the research platform.

Part III

Predictive Control in the Cloud

9

Introduction

The benefit of using COTS software and devices lies not only in the reduced up-front costs but also in development and maintenance costs. This translates into effects such as a faster time-to-market, interoperability, a large pool of competence, and iterative development. Utility computing enhances these benefits by making it possible to prototype designs at a low cost and allowing run-time demands to change without the need to replace hardware. Another benefit is that different solutions can co-exist and selectively execute in parallel or interchangeably. These solutions can also require very different hardware, memory, storage, and compute capacity. The decision of what is an acceptable, necessary, and efficient cost-to-performance ratio, and whether it can be achieved, can be evaluated in the iterative development process, changed in production, and even evaluated on-line. To allow this flexibility, cloud services, and in extension the edge and fog architectures, provide what appears as an infinite pool of resources. A design made for the cloud should make use of this abundance of resources and be capable of utilizing the different tiers of the infrastructure, from the hyperscale data center down to the edge resources.

This section presents three MPC architectures that are designed for cloud deployment and makes, from an MPC perspective, novel use of the 'infinite' compute capacity of the cloud, while providing robustness to loss of connectivity. The designs are generic and provide a controller extension through the cloud. The first chapter examines the use of parallel requests and graceful degradation, to implement a variable horizon MPC. The second chapter utilizes a hierarchy to implement three levels of overriding control signals, allowing different deployment on differently powerful nodes. The final chapter wraps a client controller in a framework that can be modeled as a setpoint governor and a feed-forward network. Explicit recovery is introduced and examined.

Before continuing, it is also necessary to define the term *quality elastic*, which is sometimes used. The term refers to the idea that a controller can modify its performance in order to be resilient or, because there is a positive

trade-off in releasing some of its resources. This does not imply that the controller is producing something of lower quality, instead, it might reduce its speed of production etc. The connection is to the use of the elastic properties of the cloud.

9.1 Related Work

Several works with a focus on a high level of abstraction and replacing existing designs with cloud counterparts have been introduced ([Pelle et al., 2019; Mubeen et al., 2017; Givvehchi et al., 2014; Hegazy and Hefeeda, 2015; Heilig et al., 2015]). Other works explicitly study the transfer of control systems to the cloud [Xia, 2012; Lyu et al., 2019; Mubeen et al., 2017; Mahmud et al., 2014; Horn and Krüger, 2016; Vick et al., 2015; Škulj et al., 2013]. These tend to completely move the controller to the cloud and focus on the cloud's performance, software frameworks, delay mitigation and monitoring. This is also what was done in Chapter 7. There are also architectures and control software that are aimed at improving the reliability of the clouds [Li et al., 2009; Nylander et al., 2018; Yfoulis and Gounaris, 2009; Saikrishna and Pasumarthy, 2016], but these also do not consider novel control design by using cloud technology.

Variable horizon MPC as introduced for nonlinear systems by [Michalska and Mayne, 1993] adds minimization of the horizon to the optimization problem in order to reach a terminal set in the shortest possible amount of steps. A variable horizon (or multiple horizon) is also used for the MPC in [Park et al., 2015], but for a different reason. In this control problem, the optimal horizon varies, and is determined by evaluating the gait of a quadruped as it prepares for jumping an obstacle. In Chapter 10, a variable horizon controller is shown as a means to implement an elastic controller, i.e., a controller that depends on available resources and response times.

Robust model predictive control is a necessity in all practical systems to handle disturbances and model errors. Three common techniques are tube MPC [Wildhagen and Allgöwer, 2020; Limón et al., 2010], stochastic MPC [Esen et al., 2015], and soft constraints [Levine and Raković, 2018]. These designs have in common that they rely on a single controller that remains online, and their primary purpose is to handle limited disturbances, not unforeseen disruptions in network connectivity and computation time. Network latency and packet drop-out can be handled in robust control by anticipation of a larger disturbance, something that generally results in a conservative controller.

One strategy for using cloud services with critical systems is to build on the concept and motivation of Simplex [Bak et al., 2009; Seto et al., 1998]. Simplex is aimed at using unverified software to achieve good performance,

without sacrificing reliability. It is tiered, with one advanced controller that provides the best performance, a reliable baseline controller, and a recovery mode that can take over in the event of controller malfunctioning. The task of the Simplex architecture is to monitor the system state, and fall back on the baseline controller if the advanced controller is at risk of executing out of specification. [Ma et al., 2020] propose Simplex for edge computing, in a design referred to as switching multitier control (SMC). In the paper, a PID is replaced by a device local LQR and a MPC executing at the other end of a wireless network. This is used to position a robotic arm. A switching logic chooses between local and remote control, based on the controlled system's state error and perceived network conditions. A data driven approach is used to train the switching logic, on a Bernoulli random distribution and a two-state Markov chain. The reliable, tiered system is shown to outperform the original PID controller.

As has previously been mentioned, the MPC is historically often used as a setpoint generator. In this case, the design relies on fast local control to bring the system into a steady state in between new setpoints generated by the MPC. The controllers are decoupled but highly dependent, and the MPC does not provide the control actions to the plant. One example of this is found in [Yang and Yang, 2007]. Assuming that the local controller does not require continuous setpoint updates, this method is naturally applicable in the cloud. However, in order to wait for the local control to settle, a large separation in frequency is required. A cascade design aimed at reducing the frequency span between the inner and outer loops is presented in [Scattolini and Colaneri, 2007; Picasso et al., 2010]. Here, robust model predictive control is used to handle the discrepancy between the performance of the high frequency inner loop, in achieving the requested setpoint, and the response predicted by the outer loop.

Another possibility is to implement gain or mode switching, common in control of non-linear systems, or due to mode changes in the plant. There are many examples of switched model predictive control, often applied to specific problems. The switching signal can be characterized as time, or state dependent and be more or less arbitrary. Switching control systems are analyzed to provide overlapping stability regions and minimum dwell-times, in order to obtain formal guarantees. [Magni et al., 2008] uses switching to improving the performance of an MPC, by splitting the state space into regions associated with different cost functions. This form is aimed at increasing the control performance through variation of the primary control objective. General overviews of controlling autonomously switching systems are given in [Ong et al., 2015] and [Zhang et al., 2016]. The latter also considers robustness, introducing a robust switching tube strategy. These papers study systems with linear models that can change more or less arbitrarily. To provide guarantees, it is necessary to compute admissible terminal sets, add

consistency constraints on the control signal, finding minimum *dwell times* (how long a system must remain in a state before a new switch happens), and formulating optimization problems that consider all admissible switching sequences.

In extension, work on distributed MPC over networks include collaborative agent systems, distributed optimizations, hierarchical control and optimal control in the presence of delays (for instance [Zheng et al., 2016; Christofides et al., 2013; Alessio and Bemporad, 2007; Lu et al., 2014; Scattolini, 2009; Stewart et al., 2010; Camponogara et al., 2002]). One example that makes an explicit case for MPC in the cloud is [Heilig et al., 2015] which presents an application for intelligent transport systems. A multi-agent system comprising many MPCs executing in the cloud is proposed conceptually, as a flexible method to handle heavy computations and large amounts of data

9.2 Research Gap

Developments in data driven methods, cloud computing, and Industry 4.0 introduce new opportunities and challenges for control system design. Future systems should support incremental development and effectively incorporate time-varying requirements, use of remote execution based on availability, and evolving software [Abdelzaher et al., 2020]. A key aspect, and a particular opportunity, is a move towards flexible and re-configurable designs [Levine and Raković, 2018; Scattolini, 2009], and providing resiliency. An emphasis on building resilient software has been one of the success factors in the latest era of computing systems [Basiri et al., 2016] and will likely be important also in industry. While in early considerations of cloud control, such as [Xia, 2015], there is an element of mitigating latency, there is a lack of work on control design that considers the possibilities and implications of the elastic, and virtually unlimited compute resources of the cloud. Rapid elasticity promises to be cost effective by quickly scaling up and down to meet demands, but distributed clouds also create a diverse execution environment. This puts new perspectives into the control design. One is to provide quality elasticity, that is, instead of requiring a certain infrastructure performance, provide the best quality that the infrastructure currently allows. Another is to keep quality constant, but reducing resource usage whenever possible. These are things that have not been explored in the control community.

A design such as plug-and-play MPC [Levine and Raković, 2018, p. 259-283], while highly related, does not consider important aspects such as allowing temporary degradation of the service. It focuses on the domain of CPS and industrial Internet, but it considers strict constraints to admit or reject components, to form a new, stringent structure with full connectivity. In distributed computing and cloud, the concept of partition tolerance, i.e., being

able to function after losing contact with the service, is important. This adds a third perspective into the design, in terms of graceful degradation.

The hierarchical design in [Scattolini and Colaneri, 2007; Picasso et al., 2010] achieves high frequency in the setpoint generator, by passing nominal requests and robustness bounds from higher to lower layers. The objective of these papers is to provide stability guarantees for the hierarchical controller, and remove the need for clearly separable slow and fast dynamics in the higher and lower layers. If the lower layer cannot meet the robustness guarantees, it resorts to a conservative baseline controller. The papers are examples of robust control, with results focused on the ability to find a robust controller on-line. There is limited consideration of some things that are important to the CCS, such as large variations in availability, the effects of computation time, and solving multiple solutions with the final decisions made locally. In [Ma et al., 2020], Simplex is used to great advantage, but the work focuses on an on-premise edge device and strictly critical control.

Controlled switching can be introduced as a performance enhancing feature. Some designs are directly transferable into the cloud context, such as the performance enhancing MPC of [Magni et al., 2008]. The switching is not critical and the design can be offloaded in the cloud, by executing the non-critical switching decision there. The MPC problem is in this case always executed on the client and the offloading creates a form of supervisory control. In cloud control systems, switching also arises due to connectivity loss or long delays. This switching is different from controlled switching. The supervisory mode also does not in itself make use of scaling features.

The various works in distributed and hierarchical control so far also largely fail to consider the cloud as a separate paradigm. In the example of the multi-agent transport system [Heilig et al., 2015], there is no consideration as to how the control structure itself can be improved using the cloud. The use of cloud in this case, and in cases including network control with strict assumptions, is an implementation detail. The performance of the cloud, as studied in Part II, and of access networks, are relevant to these cases, but the controller is not cloud specific. Similar designs should be studied, in which small, intrinsic parts of the control loop can be lifted to the cloud and performed differently in order to bridge the downsides of potential delay and connectivity issues.

9.3 General Design

The premise in the following chapters is a control loop that requires the cloud to reach its full potential, for instance due to computational restrictions on the device, or the need to access data that is not locally available. Specifically, for the sake of argument, the client exists on a computationally constrained

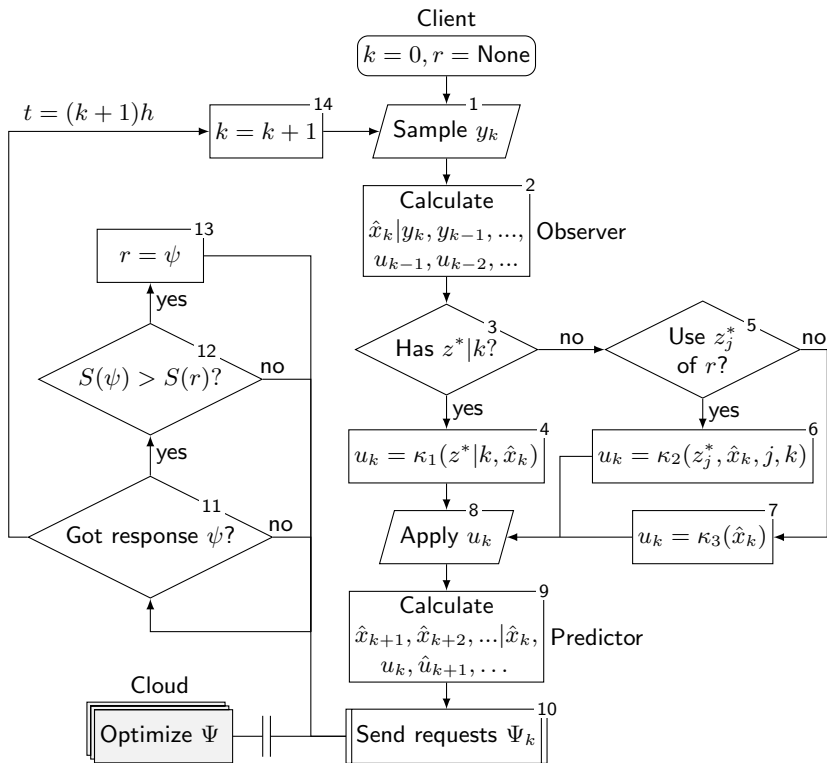


Figure 9.1 Generic flow in the client software. The function κ_1 implements closed loop predictive control. This can include robust control, such as a tube. κ_2 implements a graceful degradation to local control, and κ_3 implements the local control.

device that is adjacent to the plant.

Client

A general flow of the client control logic is illustrated in Figure 9.1. In every control period the client samples the plant sensors (1), obtains a state estimate (2), selects between three control functions (3-7), updates the plant actuators (8), predicts future states (9), sends a set of requests to the cloud (10), and collects responses (11-13). The three functions $\kappa_1, \kappa_2, \kappa_3$ represent an ordinary MPC (4), a switching procedure (6), and baseline control (7). The predictor in block 9 provides the one step prediction (i.e. $\hat{x}_{k+1} | \hat{x}_k, u_k$) introduced in Section 8.6, but can also generate other predictions, as necessary for input to the subroutine 10. The unnumbered optimization blocks (also titled Cloud) are triggered by output from block 10 and executed in parallel

with the client. Blocks (11-13) create a loop that collects responses from the cloud. The loop exits when the wall clock t reaches the next sampling time. The function S is a value function that defines an order of priority for responses from the cloud, and r stores the selected response. The decision in block 3 should be interpreted as *"if the MPC response for time k has been received"*. In the following, this will imply closed loop control and the use of $u_k^*(0)$, but it could also implement some other strategy over several control periods. The sampling time h is a base sampling rate, i.e., the shortest period the system can accept.

Constant delay

In these chapters, and as was done previously, a dead-time is forced into the controller. This is done in order to create a time frame for the optimization. A state prediction \hat{x}_{k+1} is generated using the available system model

$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k \quad (9.1)$$

This prediction is used as the initial state of the optimization at time k , and the resulting controller output is applied at the next sampling instant $k + 1$. The delay of one sample is the time frame available for the optimizations in the cloud. Any results returned later are discarded.

10

Variable Horizon Control

In a cloud implementation, the prediction horizon of an MPC can be decided dynamically with stability and feasibility enforced in the optimization. The client in this chapter implements two primary modes of operation. In assisted mode, the client uses the resources of the cloud to solve the optimization problem over a large set of horizons in parallel. In local mode, a simpler controller stabilizes the system and provides a basic degree of performance. The two modes are formed from the ordinary constrained and guaranteed stable MPC. To handle connectivity loss and extensive delays, a third mode provides the switch from assisted to local mode. The resulting system is convenient to implement, and has a built-in capacity to scale with the problem. The approach naturally extends to edge clouds, which combine the compute capacity of a centralized cloud with the low latency access of local nodes. It also puts in perspective the use of flexible, cost effective, best effort control systems as opposed to traditional, costly and static systems.

10.1 Targeted system

The system is illustrated in Figure 10.1, as a generalization of Figure 3.5. The system is composed of a plant controlled by the cloud assisted controller, here executed in the device local Client. The client continuously executes a local on-board controller κ_l and a remote controller κ_r , which performs most of its work in the cloud. Using a cloud service, the remote controller processes several potential solutions in parallel, filters the responses and forwards the best selection to a function $f_u(\kappa_l, \kappa_r)$ that merges the output of the two controllers. The function $f_x(y, u)$ provides an observer and predicts the plant state a number of sampling periods into the future. In relation to Figure 9.1 these components represent κ_3 , κ_1 , κ_2 , the observer and predictor, respectively.

This chapter is based on [Skarin et al., 2019] and [Skarin et al., 2020b]

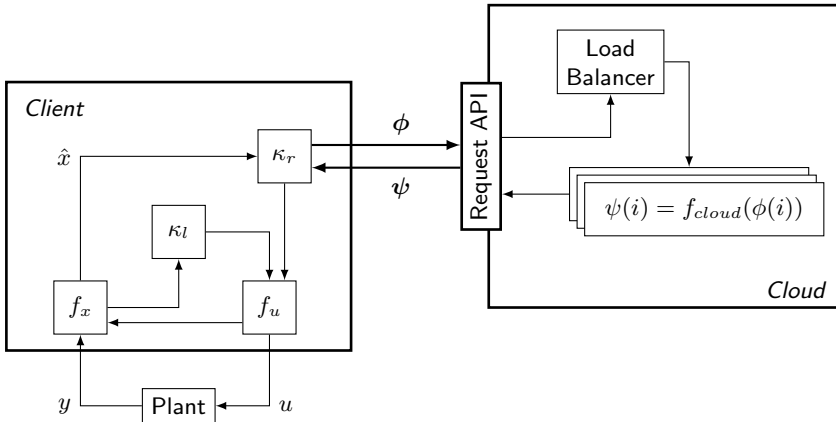


Figure 10.1 A refinement of Figure 3.5. The control law is divided into a local part $\kappa_l(\cdot)$ running in the client, and a remote part $\kappa_r(\cdot)$ that is off-loaded to the cloud. The remote part $\kappa_r(\cdot)$ is realized by running multiple parallel instances in the cloud.

A key element of the design is the use of the cloud to execute multiple instances of a model predictive controller. In Figure 10.1, this is represented by the stacked boxes labeled $\psi(i) = f_{cloud}(\phi(i))$. $\phi(i)$ represents the configuration of a request, from the request set ϕ . In the following, the variables of this request set will be the initial state x_0 and the horizon N_i , where i is the request index. This forms the input pair (x_0, N_i) . In practice, the request includes more information about the optimization, but we can assume here that $\phi(i) = (x_0, N_i)$. Similarly, the responses $\psi(i)$ must include the predicted state trajectory, \mathbf{x}_i^* , and a control input vector, \mathbf{u}_i^* , to form $\psi(i) = (\mathbf{x}_i^*, \mathbf{u}_i^*)$. In practice, this response also includes other useful information such as the processing time, the optimal control cost $V(\mathbf{x}_i^*, \mathbf{u}_i^*)$ etc.

10.2 Controller

The control strategy uses the cloud to implement a variable prediction horizon MPC. For the purpose of the work herein, it is assumed that the local device cannot execute an MPC controller locally, not even an explicit MPC ([Bemporad et al., 2002]). Instead, a LQR is used, derived from the specifications of the MPC.

The selection of the horizon in the MPC can be non-trivial. A short horizon can lead to an infeasible optimization, while a long horizon can take too long to evaluate. A scalable controller should not be bound by a predetermined horizon. The cloud allows us to run the optimization over many

horizons, and select the best alternative that provides a response within the deadline. When the state is inside the terminal set, the plant can be operated using the local controller. If the remote controller fails to provide a response, previous results are used in combination with the local controller.

Local and remote control laws

The function that executes in the cloud is an MPC, as defined in Section 4.1. The MPC includes a terminal constraint set \mathcal{X}_f and a terminal cost V_f . This ensures that the controllers defined in the individual responses are stable. A local control law, $\kappa_l(x)$, is defined such that \mathcal{X}_f is a positive invariant set for the controller, as defined in Section 4.2. The conditions on the individual MPC problems do not ensure that the switching system is stable. Arguments can be made concerning the selection of responses and switching to ensure this, but here the solution is empirically investigated. State feedback is used, assuming that the system is fully observable.

Cloud assisted controller

We now consider the cloud assisted controller. First, if a result from the optimization in Equation (4.1) can be guaranteed for all samples, then the local control κ_l is not explicitly needed. It is introduced implicitly into the MPC by adding the cost Q_f as the terminal cost, and enforcing the terminal set. A property of the cloud controller is that the availability of the MPC output is not guaranteed due to connectivity, latency or feasibility issues. Therefore there must be a local device controller available to ensure uninterrupted control of the plant. For this purpose, the stabilizing LQR from the MPC formulation is used to provide the baseline performance of the cloud assisted controller. Both controllers implement tracking, and replace \hat{x} with the state error in relation to a setpoint. For the nominal system, the local controller is optimal and does not violate constraints, as long as the state error is inside the terminal set. To ensure that the local controller is inside \mathcal{X}_f , it is necessary to use a setpoint governor to shape the setpoint, something which is returned to in the description of the *local mode*.

To implement cloud assisted control, the client sends a request set

$$\phi_k = \{\phi_k(0|N_0), \phi_k(1|N_1), \dots, \phi_k(i|N_i)\}, N_i \in \mathbb{Z}^+ \quad (10.1)$$

every time it samples the state. All requests must include everything necessary to construct Equation (4.1), but each is different with respect to the horizon. In every sampling period, the client receives a response set

$$\psi_k = \{f_{cloud}(\phi_{k-1}(i)) | t_r(\phi_{k-1}(i)) < hk\} \quad (10.2)$$

where the function t_r provides the response time of a request. As illustrated in block 12 of Figure 9.1, an ordering is applied to select one of the responses.

It is assumed that the response set only includes results that are feasible, i.e. that the solver either aborts in case of an infeasible problem, or returns an indication causing the local device to discard the result.

Equation (10.1) is restricted to a single model, cost function, and constraint set, but this can easily be extended. We can consider a request set defined as

$$\begin{aligned} \phi_k &= \{\phi_k(0|N_0, m_0, c_0), \dots, \phi_k(i|N_i, m_i, c_i)\}, \\ N_i &\in \mathbb{Z}^+, m_i \in \mathcal{M}, c_i \in \mathcal{C} \end{aligned} \quad (10.3)$$

where \mathcal{M} is a set of possible models, and \mathcal{C} a set of possible constraint configurations. This allows the parallel evaluation of combinations of horizons, models, and constraints. If m and c are further defined using a time index to form $m_i(k+j|k)$ and $c_i(k+j|k)$, this supports constraint tightening over the horizon and switched systems. It is of course also reasonable to vary the stage cost l . A further extension is to make these parameters functions that depend on the state (such as $m_i(x(k+j|k))$, $c_i(x(k+j|k))$, $l_i(x(k+j|k))$), which must then be evaluated inside the optimization. This vast potential for variations¹ motivates the use of offloading.

A note can be made also about the implementation used in the following experiments. The cloud function (f_{cloud}) allows that the client defines the components of Equation (4.26) for each request, thus supporting Equation (10.3), and varying the cost function. Equation (4.28) is constructed inside the cloud function and the model, constraints, and costs cannot vary over the horizon. Including model parameters, box constraints (a minimum and maximum for each state), and costs with each request, is of small consequence to the overhead that we saw for REST in Chapter 8.

The two modes of the client, *assisted mode* and *local mode*, and the switching strategies are now further defined.

Assisted mode

In the assisted mode, the device is connected to the network. At each sample, a new set of requests, ϕ_k , is sent to the cloud service. This set includes updated state information, and the horizons to evaluate. Up until the deadline, admissible MPC responses in ψ_k are received from the cloud. Several horizons are evaluated to increase the chance of receiving an admissible response in time.

At the start of a sampling period, the local control system selects one of the arrived responses. Assuming that responses are independent, the value function S (Figure 9.1) is applied individually to each request, providing a priority that can select or discard responses as they arrive. The selection criteria can be a simple criteria, such as always selecting the smallest or

¹Which does not stop here, solver selection can be introduced etc

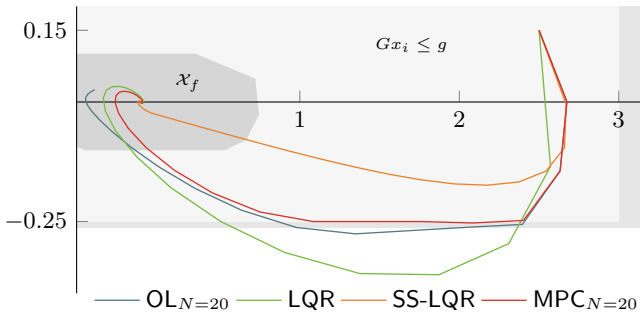


Figure 10.2 Response of the setpoint shaped LQR (SS-LQR), open loop prediction (OL), unrestricted LQR, and MPC in a system with a model error. Terminal set and tightened constraints are marked in the figure. The outer gray box shows the critical, original constraints.

largest horizon, or a more involved method which looks at the system state or the predicted cost. The selected response contains a sequence of control inputs over the length of the horizon

$$u_k^* = [u_{k|k-1}, u_{k+1|k-1}, \dots, u_{k+N_k-1|k-1}] \quad (10.4)$$

When operating in the assisted mode, and, hence, continuously receiving responses from the cloud, the controller acts as an ordinary MPC and applies to the plant only the first value in the sequence, $u_k^*(0)$.

Local mode

In local mode, the control is achieved using the LQR controller obtained from the costs and model of the MPC. Local mode is entered when connectivity is lost, the cloud is unable to provide admissible results, or when the state error lies within the terminal set. The latter implies that resources are not requested from the cloud when the state error is small. For the local controller to reliably handle setpoint changes, the perceived state error is limited. Setpoints are also restricted so that the plant operates at a safe distance from the constraints. In the following, this is referred to as *setpoint shaping*. Limiting the magnitude of the error perceived by the controller makes the local mode limited in performance, but in return provides stability and satisfies constraints.

Figure 10.2 illustrates the effect of limiting the controller in this manner. The figure shows an initial open loop prediction, an LQR, a setpoint shaped LQR, and a MPC response to a step change in the reference of a second order system. The system under control is described by the state space matrices

$$A = \begin{bmatrix} 0.9752 & 1.4544 \\ -0.0327 & 0.9315 \end{bmatrix} \quad B = \begin{bmatrix} 0.0248 \\ 0.0327 \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}^T$$

and the optimal control objective by the cost matrices

$$Q = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix} \quad R = 1.$$

An unconstrained LQR control law is derived for this problem, and a MPC is defined with state constraints $Gx_i \leq g$, input constraints $Hu_i \leq h$, a terminal set \mathcal{X}_f , and terminal cost $V_f(x) = x^T P x$. The constraints are shown in the figure, with a dark gray area representing the terminal state set, and a light gray areas showing the state constraints. In the MPC, the state constraints have been slightly contracted from the original constraints, shown as a larger, medium dark gray box. A model error is introduced in the simulation, making closed loop control necessary. The optimal path in Figure 10.2 is represented by the MPC. The figure shows that the trajectory of the setpoint shaped, conservative LQR is far from this optimal path, but also stays well within the constraints. The severity of the model error is seen in the open loop sequence, which does not correspond with the closed loop optimal path, and violates the lower constraint. The unrestricted LQR shows the reaction without setpoint shaping. This response largely violates the constraint, not due to the model error, but because of the unconstrained response.

Switching from local to assisted mode

The switch from local to assisted mode is instantaneous. When $\psi_k \neq \emptyset$, i.e. the controller has received one or more responses in time, it will use one of those results for the next control output. Similarly, if the controller is in transition from assisted to local mode, that process is immediately interrupted.

Switching from assisted to local mode

The switch to local mode can happen when the system has entered into the invariant set of the local controller, or because no response was received from the network in time. This switching is critical, since the local control does not handle constraints. In the first scenario, when having entered into the invariant set, it is straight forward to hand over control to the local controller. Being in the invariant set ensures that the local controller can act without violating the constraints. In the second scenario, connectivity is lost or the set of admissible responses becomes empty. If local mode is entered immediately when this happens, the system can experience erratic behavior or large constraint violations, because of the limitations of the local controller. To avoid this, the control sequence from the latest selected MPC response is used to provide a sequence of control inputs used in combination with the local controller.

The client controller can be applied to the prediction error as

$$f_u(\kappa_l, \kappa_r) = u_i^*(k-i) - K\hat{x}_k|x_{i-1} + K\hat{x}_k|x_{k-1}, \quad (10.5)$$

where i is the time step in which the MPC with output sequence u_k^* was selected. The state estimate $\hat{x}_k|x_{i-1}$ is the prediction of the current state used by the the MPC to generate u_k^* , while $\hat{x}_k|x_{k-1}$ is the current state prediction from the previous sample, i.e., using the latest information as opposed to the information available when the MPC was requested. The expression in (10.5) removes the predicted LQR response from the MPC output, and re-adds the LQR using updated state information. In the first step, when $i = k$, the two last terms cancel and the closed loop MPC of Equation (4.1) is obtained. In assisted mode, this happens repeatedly, as $i = k$ when the response set is non-empty, $\psi \neq \emptyset$. When $i < k$, the states may not match, and the LQR, working in closed loop, is allowed to compensate for prediction errors. This strategy assumes that the MPC, in addition to the control signal sequence, also returns the corresponding state sequence, i.e., that this is provided in the response set ψ_k . Notice that Equation (10.5) is defined to match exactly the MPC when there is no request loss.

Using (10.5), the local LQR is active at all times, and applied as a residual to the MPC open loop trajectories during the mode switch. As a further measure to create robustness and a smooth transition to local control, (10.5) is extended with an averaging term, α_i , which gradually decreases the impact of the MPC control signal when switching from assisted to local mode. The local controller is also using setpoint shaping, and the notation \hat{x}_k^{lim} is introduced to make this explicit, denoting the limited state after shaping. The α -switching control law becomes

$$f_u(\kappa_l, \kappa_r) = \alpha_{k-i}(u_i^*(k-i) - K\hat{x}_k^{lim}|x_{i-1}) + K\hat{x}_k^{lim}|x_{k-1}, \quad (10.6)$$

where α starts at one and decrease to zero,

$$\alpha_0 = 1, \alpha_{k-i} \rightarrow 0 \text{ when } k-i \rightarrow N. \quad (10.7)$$

Equation (10.6) is implemented in the client (inside f_u in Figure 10.1), and is not part of the MPC problem. Figure 10.3 shows again the scenario in Figure 10.2, but this time with two examples of α -switching. The sequence of α values is defined as

$$\alpha_i = 1 - \beta_i, \quad i = \{1, \dots, N\}, \quad (10.8)$$

$$\beta_i = \beta_{i-1}e^\gamma, \quad \beta_0 = 0.01, \quad \gamma = \log(99)/N, \quad (10.9)$$

where the exponential decay is chosen from the observation that model errors grow exponentially.

The two switching trajectories in Figure 10.3 have suffered connection loss, and are switching to the local mode using the exponential decay given by Equation (10.8), with different values of N . There are several things to note in the figure. First, the closed loop MPC (green) represents the assisted

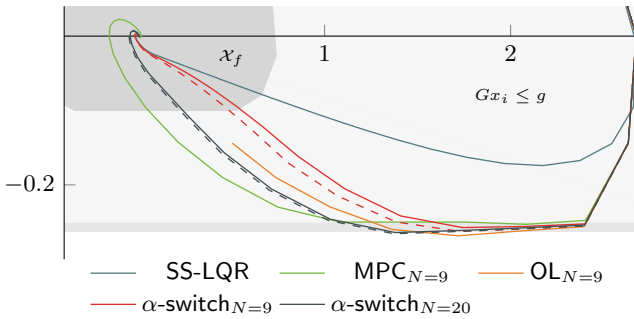


Figure 10.3 Same scenario as in Figure 10.2, with different controllers. The local, setpoint shaped LQR (SS-LQR), a fully connected MPC_{N=9}, initial open loop prediction and two, disconnected, α -switching controllers, using (10.6) with the decay in (10.8). Dashed lines show the paths for a fixed $\alpha_i = 1$, i.e. (10.5).

mode. This is the typical path of the cloud assisted MPC controller, with a horizon of nine steps. Second, the α -switch with $N = 20$ (gray) violates the original constraints, while the α -switch with $N = 9$ (red) does not. To handle connectivity issues, it is a more robust strategy to select short horizons, since they are forced into the terminal set within fewer steps. Third, the two dashed lines shows paths that use $\alpha_i = 1$. This is to be compared with the solid lines using the α -switch. The effect of α -switching is clearly visible when $N = 9$. The effect is reduced as N increases, since a large part of the horizon is inside the terminal set. Fourth, the α -switching initially pulls the path inwards towards the setpoint shaped LQR (blue), but as the system approaches the terminal set, the $N = 9$ solid path again begins to merge with its corresponding dashed path. This happens even though the local controller becomes more and more dominant, because the paths of the two modes begin to coincide. The strategy moves the path towards SS-LQR, with the result that the slightly contracted constraints allows the path of $N = 9$ to stay within the original constraints. Finally, on close inspection, the path of the closed loop MPC temporarily violates the constraints at one point. This can happen close to the constraints. When this happens, the MPC becomes infeasible, but the client easily recovers by temporarily entering the switching mode. The switching brings the state back inside the constraints, where in turn the assisted mode, and therefore MPC efficiency, is regained.

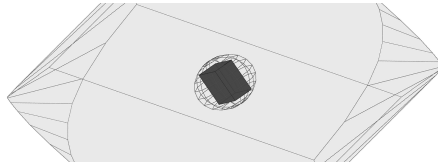


Figure 10.4 The maximal invariant set (outer hull), an invariant ellipse, and the contained invariant set cube represented by three linear constraints. The local controller is constrained to operate in this limited space.

10.3 Evaluation

This section provides an evaluation of the assisted controller, using the reference plant from Chapter 5, simulations in Matlab, and data from the cloud platforms in Part II. Results are presented after some notes on deriving the assisted controller, and on the simulation setup.

Deriving the assisted controller

The offloaded function that is executed in the cloud is the linear quadratic optimization in Equation (4.26). This is used to implement the remote MPC, represented by κ_r in Figure 10.1. Since the selected MPC is a linear quadratic problem, an unconstrained linear state feedback controller $\kappa_l(x) = Kx$ is derived directly from it, and used for the local mode. As specified in Section 4.4, the terminal cost follows as the asymptotic cost in the LQ problem. This procedure was presented in Section 4.4. The model matrices A , B , the constraint pairs (C_x, c_x) , (C_u, c_u) , cost matrices Q , R , Q_f , and the terminal set \mathcal{X}_f must be specified. A strategy must also be defined for choosing the request set ϕ_k . Here, this strategy consists of using a fixed set of horizons, which are evaluated every sample. These horizons are listed with the results in Tables 10.1 and 10.2.

Finally, a terminal set that is invariant for the local mode controller must be defined. Based on the assumptions and purpose for creating the cloud controller, it is not necessary to find the largest possible invariant set, a reasonably small and robust subset will suffice. Polytopes can represent the invariant set arbitrarily well, but a general polytope can become very complex and hard to calculate. An alternative method to find a reasonable non-optimal subset is to use elliptical constraints. However, this requires the use of cone programming for the optimization. To provide the linear constraints for the QP problem, a rectangular polytope

$$X_f = \{x | H_t x \leq h\} \quad (10.10)$$

is defined, which fits inside the elliptical subset. This is illustrated in Fig-

ure 10.4². The outer hull in this figure is the maximal invariant set for the LQ controller. The ellipsoid is a simple sphere that fits inside this invariant set, and the small cube inside the sphere is the final terminal constraints. Naturally, this provides a very pessimistic terminal set, but also one that is easy to find and represent. If this set is later updated and grows, it will reduce the necessary controller horizon which can reduce the computational load on the cloud. This in turn can translate to higher response rates, improving the controller response, and reducing monetary costs. A thorough introduction to the use and calculation of invariant sets is found in [Blanchini, 1999].

Finally, the setpoint must be shaped to ensure the invariant property of the local controller. A *governor* function is applied to limit the perceived error of the tracked state variable x_1 , so that

$$x_e^{lim} = [x_e^{lim}(1) \quad x_e(2) \quad x_e(3)]^T \quad (10.11)$$

where $x_e = x - x_{sp}$ is the state error, and x_{sp} the setpoint vector. A simple procedure is to limit x_1 , using the terminal constraint.

Simulation

The simulations use Matlab, and Simulink, with a continuous-time model of the plant, as shown in Figure 5.2. Matlab's *quadprog* is used for the optimizations, with support for the terminal conditions. However, as explained in the following, another optimizer is used to obtain a processing time for the request.

For each individual request, two sources of delays are considered, and the latency of a single optimization, i.e., of one configuration $\phi_k(i|N_i)$, is modeled as

$$\tau_{rt}(N, \epsilon) = X_p(N, \epsilon) + X_s \quad (10.12)$$

where τ_{rt} is the round trip delay, and X_p and X_s are random variables referred to as the processing time and service delay. The processing time depends on the controller horizon and the current state error ϵ . A repeated request is expected to not provide an identical processing time, due to executing on different machines, and alongside other applications in the cloud. The function and variability governing the processing time is unknown. To obtain an estimate of the processing times as a function of state and horizon, and variability due to the cloud, an efficient optimizer (QPgen, [Giselsson, 2015]) is deployed in a cloud service. A large set of measurements using this optimizer is used as input to create distributions for $X_p(N, \epsilon)$ and X_s . However, this optimizer lacks the necessary support for terminal conditions, and therefore, the final evaluations are simulated in Matlab.

²Using the Multi-Parametric Toolbox by [Herceg et al., 2013].

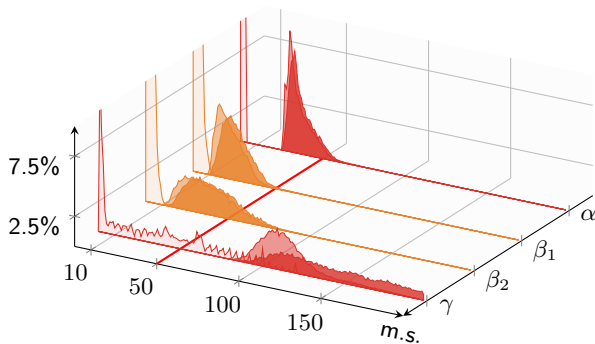


Figure 10.5 Delay distributions from three different services. The least transparent, darkest, distribution in each configuration is the round-trip time. The medium transparent is service time, and the most transparent, lightest, distribution is the processing time. β_2 is loaded by twice as many parallel requests as the other configurations. This is also compensated by more workers.

The service delay includes networking and other overhead such as queuing for access to the cloud service. It is assumed that the distributions are relatively stable, and therefore, they do not depend on the time step k . In practice, if the load on the system temporarily causes the distributions to change, the resulting request loss is managed by the switching and local modes. X_s can be calculated through measurements of $\tau_{rt}(N, \epsilon)$ and $X_p(N, \epsilon)$ when running QPgen optimizations on real cloud services.

The processing time's dependence on the state error ϵ and horizon N was shown in Figure 8.2 and Figure 8.3. These figures show results from the following experiments, using QPgen executed in the cloud. The experiments are limited by the lack of terminal constraints, but are comparable in relative terms, and assumed to be sufficiently representative of the optimal controller executing in the cloud. This can also be seen from the results in Chapter 8.

Processing time and service delay

To obtain input for the simulations, measurements were performed on HAProxy configurations and Lambda, as introduced in Chapter 8. Compared to the measurements in Chapter 8, this evaluation uses a different implementation for the optimization, and executes parallel requests (towards the same cloud) with every sample.

Figure 10.5 shows results from executing the MPCs on three different platforms. These platforms are a HAProxy cluster on AWS EC2 (α), a HAProxy cluster in ERDC (β_1, β_2), and AWS Lambda (γ). As usual, while the details of clusters change, the remote request API, the code of the remote function,

and the implementation on the client remain unchanged. The tests repeatedly evaluate a step change in the setpoint for the reference system (Chapter 5) with constraints $|x_1| \leq 0.55, |x_2| \leq 1.5, |x_3| \leq \pi/4$. For a set of horizons $N = 5 - 120$, the step is evaluated a total of 1000 times, using each service. The services γ , α , and β_1 are continuously loaded with ten optimizations in parallel, while β_2 must handle twice as many, i.e. 20, parallel requests. With each new request, the horizon is selected randomly from the above set.

The public FaaS, γ , uses the default service provider configuration for the function, in terms of the container environment, concurrent requests, available memory, maximum processing time etc. This default configuration of the service is not restrictive for the scenario. Examples α , β_1 , and β_2 use the infrastructure service of the cloud (IaaS) to create a custom service. This uses Python Flask and HAProxy to provide the request API and a load balancer. The data center provider and virtual machine configuration are different for α and β_1 , but both have one load balancer and the same number of total worker CPUs (eight). β_2 is provided more cores to account for the additional load.

The aim is to use a sampling period of 50ms (shown as a red line across the experiments in Figure 10.5). This is not possible for γ , since service times, and sometimes also processing times, end up beyond the red line. The large spread of the processing times, $\mathcal{X}_e(N, \epsilon)$, seems due to that the service penalizes extended executions. This is also what was seen in Chapter 8. That is, the performance per time unit decreases when the size of the problem or the required iterations in the optimization increases. This translates to long processing times for large horizons, and when the system state requires many iterations for the optimization to converge.

The results for γ prompt the necessity to try something different, to find a distribution that works with the controller. This is why α and β_1 were introduced. The response of α is much better than that of γ . Since they use the same cloud provider, with the same locality, this shows that the long delays of γ , caused by processing times, are due to the performance of the public FaaS, not the general performance of the cloud. This is again in line with the results in Chapter 8. The figure also shows that using the local cloud provider, β_1 , gives a response similar to α . The more distant provider has faster processing times, but a larger minimum service time. This is attributed to network delay. Finally, β_2 loads the ERDC cluster with more concurrent requests, to allow for more horizons per sample. Although the number of workers are scaled up to account for the load, the service time increases. The ideal cloud scenario of servicing an arbitrary amount of concurrent requests, without affecting the individual latency, does not fully hold for this configuration and load. Still, if counting only the number of successful requests per sample, this setup has a higher utility than β_1 .

Because β_2 can serve more requests and includes a reasonable amount

of failed requests, it is chosen as the baseline to study control performance. These experiments were executed from a workstation located at the plant in Figure 7.1. We know from Part II that the overhead of the network transmission to the local data center is generally small, and are independent of the plant and client state. Because there is a sufficient number of workers in the cluster, we also assume that admission times are independent of the plant and client state. The service time distribution, X_s , is obtained through maximum likelihood fitting of the data in Figure 10.5 onto a log-normal distribution. Processing times are more complex, since they depend on the plant state and the horizon. Figure 8.2 and Figure 8.3 show this, but do not provide the full picture, since they only illustrate the horizon and variations in state x_1 . Each point is an average of another grid of a few values for state x_2 and x_3 . Instead of creating theoretical distributions for the processing time, the values of $\mathcal{X}_e(N, \epsilon)$ are obtained by executing the functions on the cloud service, as part of the simulation. As seen in Figure 10.5, the processing times do not change between β_1 and β_2 , because the clusters are not overloaded. The simulations limit the number of parallel requests to retain this property. Still, there may be differences to the results in Figure 10.5 in terms of longer processing, due to other tenants etc.

Results

The performance of a nominal system is shown in Figure 10.6, Figure 10.7, and Figure 10.8. The simulations are run in Matlab, but draw the processing delays from the cloud, as presented in the previous section. In examples A and B, the service times are drawn from a log-normal distribution,

$$f(x) = \frac{1}{x\sigma\sqrt{(2\pi)}} e^{-(\ln x - \mu)^2 / 2\sigma^2}, \quad (10.13)$$

where $f(x)$ is the probability density function. The used parameters are $\mu = 2.54$, $\sigma = 0.48$, and an offset of 6.8 is also applied. This represents the service of β_2 in the previous section. For examples C and D, problematic network/service conditions are simulated by increasing to $\mu = 3.93$.

Examples A and B in Figure 10.6 illustrate two different controller modes. In A (left), S is defined to select the longest horizon from the admissible set ψ_k . In B (right), S is instead defined to select the shortest available horizon. Table 10.1 shows the percentage of times each horizon provides an admissible result, i.e. is feasible and responds in time. In examples A and B, service conditions are good and most requests respond in time. Looking at Table 10.1, the decreasing values for horizons 11 to 3 are not due to service or processing times, but because the horizons are too short to provide feasible solutions. Horizons from 20 and above always provide feasible solutions, but a few responses are lost due to delays. The lower values for the longer horizons are attributed to processing time delays.

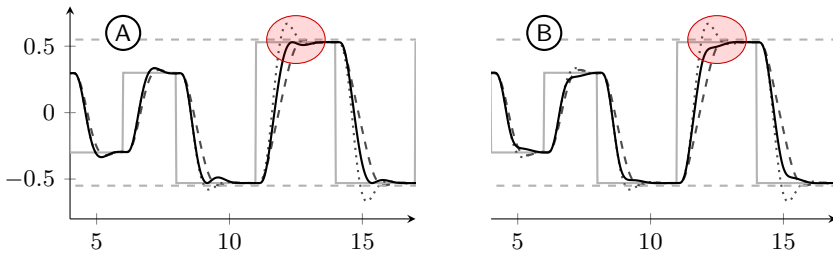


Figure 10.6 (A): Assisted controller selecting the longest horizon. (B): Assisted mode controller selecting the shortest horizon. The gray trajectory is the setpoint and the state constraint is shown as the dashed gray line. A dotted gray line shows the unconstrained LQR, and the dashed black line show an LQR with setpoint shaping to not violate the constraints.

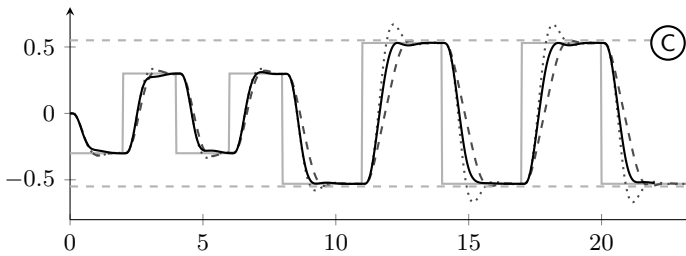


Figure 10.7 The scenario in Figure 10.6-B with degraded service.

Table 10.2 shows how often each horizon is selected by the controller. Here, $N = 0$ refers to the local mode of operation. Since the service performance in β_2 is good, selecting the long horizon effectively translates to using $N = 120$ or $N = 111$ almost always, i.e., this system behaves similar to a standard MPC with a long constant horizon. Service latency is the reason why $N = 120$ is not always selected in example A. In B however, the selected horizon will decrease as the state error gets smaller, and the controller therefore operates over a range of horizons by design. In example A, the local mode was almost never used, since the system does not have time to stabilize around the setpoint before it changes. In difference to A, example B is able to often execute in the local mode. This is due to the use of shorter horizons. The difference in behavior between the two modes is clearly visible as the system approaches a setpoint, emphasized by the encircled red areas in the figure. Both systems stay within the constraints, which is not the case for the unconstrained LQR, shown in the dotted line. Both are more efficient than the setpoint-shaped LQR, shown as the dashed line.

Example C in Figure 10.7 also uses short horizon selection but the ser-

Table 10.1 Admissible responses (percent)

N	3	5	7	9	11	20	29	38	47	56	66	75	84	93	102	111	120
A	26	40	51	60	69	94	94	96	94	92	90	93	91	92	87	90	85
B	23	36	47	56	65	95	98	98	99	99	96	95	92	95	94	91	88
C	7	13	19	20	28	39	25	31	26	27	24	21	24	20	23	16	19
D	6	8	11	12	12	17	22	21	17	15	15	14	10	4	11	8	9

Table 10.2 Horizon selection (percent)

N	0	3	5	7	9	11	20	29	38	47	56	66	75	84	93	102	111	120
A	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	24	73
B	55	15	5	4	4	4	11	2	0	0	0	0	0	0	0	0	0	0
C	40	4	10	11	6	8	9	3	4	2	1	1	1	1	0	0	0	0
D	20	5	5	5	5	2	11	10	15	7	4	2	3	2	0	1	1	0

vice delays are now longer. In Table 10.1 there is a clear decline in admissible responses, as seen in the lower numbers and reduced blue color. The effect of combined service time, processing time and feasible horizon also becomes clearer with a peak of admissible responses at horizon 20. In Table 10.2, the use of horizons above 20 shows that on occasion, the network conditions of C causes unnecessarily long horizons to be the only choice available. This translates into some of the used trajectories resembling example A, and some example B. Overall, the controller continues to perform well. For the switching strategy to be reliable, it is useful that the MPC moves quickly away from constraints and the shaped local controller is switched in early. When using the mode switching, selecting the shortest feasible horizon creates these conditions, but due to service latency and potential execution variability, the shortest feasible result may not arrive to enter the admissible set.

In example D, Figure 10.8, a change has been made to the controller constraints. The range of state x_1 has been more than doubled, which allows for a larger range of setpoints. At time $t = 5$, this is used to request larger state changes than what was previously possible. This example also uses short horizon selection. The larger setpoint changes may require longer horizons for feasible solutions. Up until now, the controller could limit itself to use horizons 3 to 20. When it now needs longer horizons, they may not be admissible due to delay. If they become available, they can be used, and if they do not, the local mode will eventually bring the system to a state where the shorter horizons are feasible. The larger setpoint in D is observed in Table 10.2, as the increased selection of horizons above 20. The peak in admissible responses is not as clear as in example C, but there seems to be

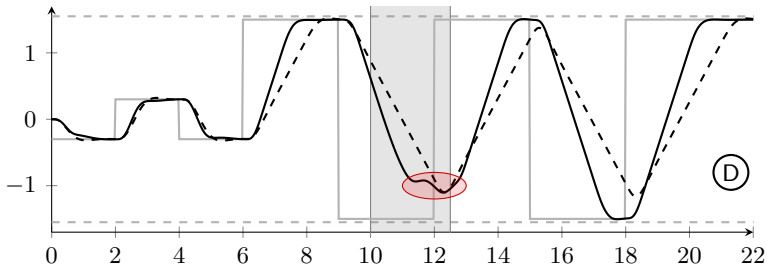


Figure 10.8 Assisted control with degraded network and modified constraint. In the gray area, the controller has no connectivity to the cloud and transitions to local mode. Performance is regained when connectivity is restored. Dashed line is a setpoint shaped LQR.

a slight shift upwards in Table 10.1, which is expected with the larger setpoint changes. In addition to modified constraints and the degraded service conditions, example C also introduces connectivity loss at $t = 10$. When this happens, the controller must switch from assisted to local mode, using the open loop data of the latest selected MPC. After connectivity is restored, the controller enters the assisted mode again as soon as a feasible MPC is returned in time. The encircled red area shows the switch to and from local mode. It may seem that the trajectory leaves the optimal path prematurely, but this is part of the α -switching and short horizon strategy, which prioritizes robustness over the potential performance of the open loop data from the MPC.

10.4 Edge perspective

After seeing how the variable horizon controller works, we now briefly consider an application in the distributed cloud. Consider the setup in Figure 3.5, where the client is connected to two systems, both of which can provide offloading. One is a large cloud that provide virtually endless and cheap computations, but also introduce largely random access times. The other is an edge node with limited and costly resources, but with access times that can be considered constant. For the purpose of the following example, we assume two ideal properties from the cloud. First, the network and admission delays are independently drawn from a single distribution, and second, executions are not interrupted, resulting in processing times that depend only on the number of iterations in the optimizer and the size of the control problem. This is an idealization of the cloud performance, which, as we saw in Part II, cannot be assumed in practice. It is nonetheless conceptually useful and illustrative.

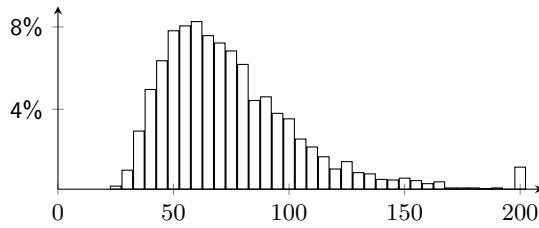


Figure 10.9 Probability density of delays drawn from a log-normal distribution with $\mu = 4$, $\rho = 0.5$ and an offset of 14 ms.

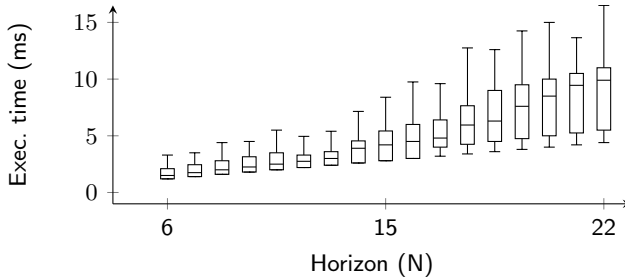


Figure 10.10 Experienced execution time distributions for various control horizons when executing the experiments. A box shows the (lower) 0.25-quartile to the (upper) 0.75-quartile. The line inside the box is the median. Whiskers show the lower quartile $-1.5 \cdot IQR$ and upper quartile $+1.5 \cdot IQR$, where IQR is the interquartile range (the difference between upper and lower quartiles). Outliers beyond the whiskers are excluded.

The networking and admission delays towards the larger cloud are represented by the log-normal distribution (Equation (10.13)), with parameters and a normalized histogram over a large set of samples illustrated in Figure 10.9. This distribution was originally obtained from measurements on a public FaaS. Box plots of execution times are shown in Figure 10.10. This figure shows one box plot per available horizon and the variation in execution time comes from the number of iterations required in the optimizer for different plant states. Outliers are excluded from the figure and there are instances of the larger horizons that execute well beyond fifteen milliseconds. Notice that even with the idealization of the cloud, there are unknowns in the number of iterations and the access times.

The response deadline is again 50 ms, meaning that a majority of requests will fail already due to the access times in Figure 10.9. However, because the response times are independent, multiple requests will increase the chance of a timely response. We can further increase the chance by allowing shorter

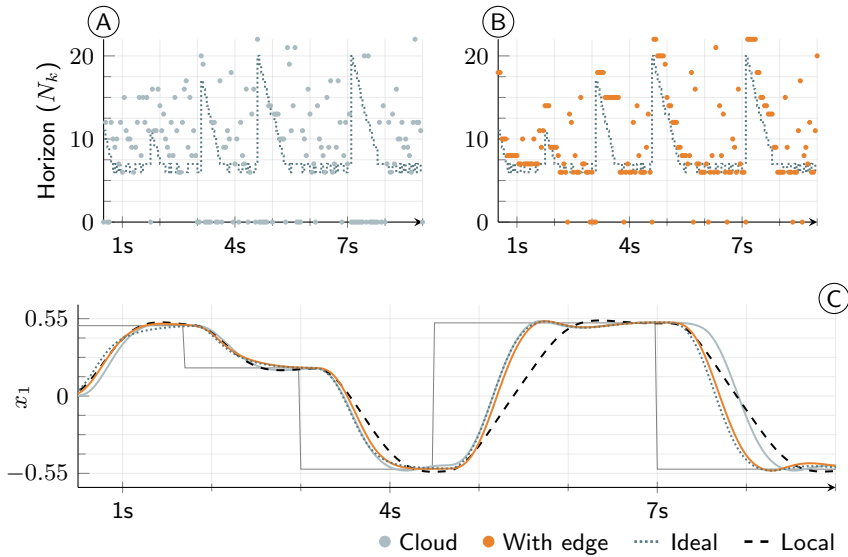


Figure 10.11 The time lines have been cut for illustrative purposes.

horizons, since they execute faster. However, only using short horizons will be insufficient, since they will become infeasible as we move away from the terminal set.

Now turn to Figure 10.11. This figure shows the results from an experiment with the theoretical data center cloud, and results from using an edge cloud. We return to the inclusion of the edge cloud shortly. The upper left plot (A) shows the used horizons when executing the controller, using the distribution in Figure 10.9 and the execution times in Figure 10.10. The dotted line shows the response of an ideal scenario, where all the horizons are evaluated without network, admission or processing time delay. While not completely random, the large delay causes the used horizons to be spread out, and only 75% of the iterations in the controller provide feasible results. Plot C at the bottom shows the closed loop response in state x_1 , i.e., the position on the beam. The plant simulation is setup as presented previously in the chapter, with one difference. Local control is only used as a backup, with MPC responses used until they are exhausted, i.e. applying Equation (10.6), also when the plant is inside the terminal set.

The response without a networked controller is shown for reference as a dashed line, and the response with no delay is included as the dotted line. On occasion, the closed loop response of the cloud controller deviates from the ideal, but the seemingly *random* horizon and recurrent lack of responses from the cloud do not cause any large build up of errors. One very noticeable

degradation is at the step change after seven seconds. At this point the client experiences a large amount of request loss in combination with a setpoint change. The reason for this behavior is that the setpoint of the client only changes when responses arrive from the network. This could be implemented differently, and with setpoint changes allowed during local/switching mode, the client would start to move the trajectory towards the new setpoint.

It is assumed that the edge node only can handle a single request at a time, but that it provides a constant access latency. Execution times are the same in the edge node, i.e., Figure 10.10. The latency of the edge node is set to 40 ms. This is substantial, and longer than the shortest delays in Figure 10.9, but also much shorter than the mean and median of the cloud distribution. This means that the edge can service most, but not all, requests when the horizon increases. The edge is resource constrained, and assumed costly, so the load should be kept low if possible. This makes for two reasons to keep the horizon short. One, is to increase the chance of a successful response, and the other to keep resource usage low.

Assuming no prior knowledge of the delay and processing times, and no knowledge about whether a new configuration of the controller will be feasible, the following strategy is selected. Assume that the controller used at time k is feasible also at time $k + 1$. Let the edge execute the latest verified configuration. If this configuration becomes infeasible, replace the edge configuration with a controller that is likely to solve the problem. For simplicity, the edge is re-configured for the longest horizon if the controller becomes infeasible. Looking at the results in figure Figure 10.11, this strategy regains a lot of structure in the used horizons, and since the edge can respond to most of the request in time, there are very few occasions where open loop and local control must be used. In fact, most of these occasions happen when the ideal controller switches between two short horizons. This is due to some unforeseen corner case in the optimization, and the reason why the edge solution can experience a largely increased horizon even when there is no setpoint change. The response in Figure 10.11-C closely resembles the ideal.

What should be noted from this experiment is that the used strategy is very generic, assumes no knowledge about the properties of the distributed cloud. Still, the setup has several useful properties. Computations at the edge are kept low, by continuously updating the controller to fit the problem. Meanwhile, the edge configurations are only updated after being verified in the larger cloud. There are several redundancies in the system. The edge ensures almost certain closed loop control. If the edge controller fails due to unforeseen reasons, there are two recovery systems in place. The larger cloud provides redundancy for the constrained MPC. The client's gradual degradation to local mode provides a second level of recovery. If the edge is removed, the MPC controller will continue to function, and if the cloud is removed, local control brings the system into a low performance mode.

10.5 Conclusion

The chapter has presented the implementation of a variable horizon model predictive controller using the cloud. The performance of the cloud services and the controller were shown in a combination of benchmarks, performed on the real services in the cloud, and controller simulations driven by the observed data. Through a combination of local LQR and cloud supported MPC, an improved MPC design was obtained, which allows flexible performance. In the evaluations, the controller did not show indications of becoming unstable. Constraint satisfaction is made possible in the nominal case, and a strategy combining short horizon selection and gradual mode switching provides smooth operation, performance and a degree of robustness. The idea of elastic control extends to complex and non-linear systems, and the strategy of evaluating many controllers concurrently should lend itself to many problems for finding cost-minimizing solutions online, and for creating robust best-effort solutions in control.

The variable horizon used in this chapter, and the exemplified extension to an edge cloud environment, are crude approaches used to investigate and illustrate the concept of quality elastic control. In practice, components such as the request selection should rely on heuristics to be discriminating, and change the selection over time. For instance, if an unanticipated change in the state or control cost happens, the selection can shift or grow the range of horizons, and increase the number of requests sent to the cloud. Another example, to replace the shortest horizon principle, is to compare the cost of each feasible trajectory and remove long trajectories that do not sufficiently reduce the cost. Similar results may apply if the selection takes the longest horizon, where $x(N - 1|k) \notin \mathcal{X}_f$, i.e. the longest horizon that reaches the terminal set in the final state. Importantly, the resilient, quality-elastic, cloud controller provides a general applicability of such heuristics based on context, and the potential for evolving them online.

11

Rate Switching

This chapter looks at a two-tier architecture, consisting of a high rate MPC in the cloud and a low rate MPC on the local device. The MPC in the cloud is the nominal controller. The system switches to the local MPC in case of an unresponsive network. The two MPCs are designed to be as similar to each other as possible, except for the sampling rate. The evaluation considers different alternatives for when to execute the local MPC and how to perform the switching.

From a control analysis point of view, the obtained system is a non-linear switched system. A helpful property is that the physical plant under control never switches. It is only the controller that switches, in particular the sampling period and the equality constraint corresponding to the ZOH-sampled process model. A challenge is that the switch from the cloud MPC to the local client MPC is completely arbitrary, and it is not, e.g., possible to impose any dwell-time constraints. The switch from the local client MPC to the cloud MPC can, however, be delayed if necessary. The interest is in observing the result of a well-defined controller, to see if it appears problem-free. The hypothesis is that there will be no really problematic, unanticipated dynamics, because of the small differences between the controllers.

11.1 Targeted system

The control architecture consists of two layers; A local device layer located close to the plant, and a cloud layer that executes somewhere in the cloud, see Figure 11.1. The local device performs the sampling, state estimation, sends the data to the cloud MPC, and actuates the returned control signal. The client is not able to execute the same MPC problem as the cloud, but is able to evaluate a less computationally complex and, hence, faster to solve, MPC problem. This is in contrast to Chapter 10, where it was assumed

This chapter is based on [Skarin et al., 2020a]

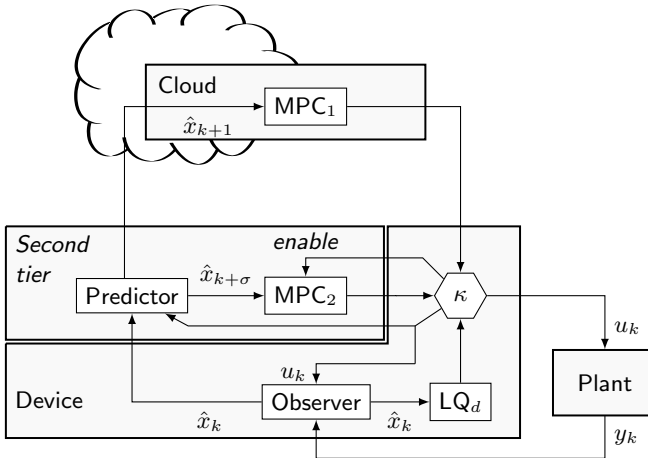


Figure 11.1 Setup of a local constrained controller, MPC_2 assisted by a high frequency remote, MPC_1 , and a local unconstrained controller, LQ_d , for fallback recovery.

that the local device only had sufficient resources to execute a 'conventional' controller, in that case a LQR. There is still an LQR on the device in Figure 11.1, here referred to as LQ_d . There is also a component on the device, referred to as the *Second tier*, which implements an MPC. The Second tier and the Device in Figure 11.1 together create the client in this chapter. The Second tier is drawn as a component, since it can easily be moved to some other location in a fog or cloud network, for instance, to the LuMaMi RBS break-out in Chapter 7 (see Figure 7.1). It is assumed that communication is reliable and without delay between the Second tier, the Device, and the plant. It is only MPC_1 in the figure to which communication is unreliable.

There are several ways of obtaining a simpler problem for MPC_2 compared to MPC_1 , e.g., using a reduced order model, or by only using the MPC for a smaller physical part of the plant. Here, a less time consuming problem is obtained by solving the original problem less often. The control frequency is reduced on the client. Then, by keeping the same temporal horizon, N_s , in both MPCs, the horizon in terms of number of samples, N , is decreased in the local MPC. Hence, using the proposed approach, the optimization problem both becomes smaller, and the time available for solving it increases. The decrease in sampling rate is selected so that both the fast and the slow rate give acceptable performance. The meaning of $\hat{x}_{k+\sigma}$ and the *enable* signal in Figure 11.1 relate to the sampling rates and the interaction between the controllers MPC_1 and MPC_2 , and are explained in the following.

11.2 Controller

Execution time and sampling rates

There are several rules of thumb for selecting the sampling period of a discrete-time controller. In most cases, they are based on the properties of the closed loop system. Equation (11.1) is an example of one such rule of thumb [Xu, 2017].

$$0.15 \leq w_b/f \leq 0.6, \quad (11.1)$$

where f is the sampling frequency in Hz, and w_b the closed loop bandwidth in rad/s. This rule is based on the allowed change in the phase margin, as a result of the sampling, and it specifies an interval of acceptable sampling rates. As long as the sampling rate lies in this interval, the controller will have reasonable performance, while not requiring an excessive resource utilization. Equation (11.1) puts the frequency multiplier between the two controllers in the range 1 to 4.

Setup

The controller architecture was shown in Figure 11.1. It consists of a remote MPC executing in the cloud, with a sampling period of T_s , and a local client MPC, with a sampling period of σT_s , where $\sigma \in \mathbb{N}^*$, i.e. is a non-zero positive integer. A LQR is used as a backup in case also the local MPC fails to solve the optimization problem. The device controls a plant, and has an arbitration mechanism in κ to select the current control signal.

The plant is sampled at the base frequency, which corresponds to the sampling period of the cloud MPC, i.e., T_s . The estimator generates a prediction of the state at time $(k+1)T_s$ and $(k+\sigma)T_s$, i.e., a prediction for the fast MPC and the slow MPC. Input to the local LQR is produced by the observer. The arbitration mechanism in κ selects the final control signal. It is assumed that the execution times of the LQR and the arbitration are negligible.

Controller synthesis

The starting point for the control synthesis is the continuous-time plant model

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (11.2)$$

where $A \in \mathbb{R}^{n \times n}$ and $B \in \mathbb{R}^{n \times m}$. This system is ZOH-sampled to yield two discrete-time systems

$$x((k+\sigma)T_s) = A_\sigma x(kT_s) + B_\sigma u(kT_s) \quad (11.3)$$

where the sampling period multiplier, σ , is used to index the models. $\sigma = 1$ is the *high frequency* model, with sampling period h , used in the cloud MPC.

$\sigma = 2$, i.e., a sampling period of $2T_s$, is used for the *low frequency* mode in the client. In the remainder of this chapter, these will be respectively referred to as MPC₁ and MPC₂, with the subscripts matching the frequency multiplier.

The continuous-time infinite-horizon LQR for the system (11.2) is given by the solution to

$$\underset{u}{\text{minimize}} J(x, u) = \int_{-\infty}^{\infty} x(t)^T Q_c x(t) + u(t)^T R_c u(t) dt \quad (11.4)$$

with Q_c, R_c symmetric and positive definite. The derived controller is referred to as LQ_c. The discrete version of this controller, sampled using the base frequency (sampling period T_s), is referred to as LQ_d (with gain K_d), and is used as the backup controller on the local device.

The specification of LQ_c is also the basis for the design of the two MPCs. The goal is that the two controllers should be as equivalent as possible, except for the sampling periods. Therefore, the cost functions are derived as the discretized versions of the continuous-time cost function of LQ_c.

The discretized version of the continuous-time cost for the system (11.2) is given by

$$V(t, t_k) = x_k^T Q_\sigma x_k + 2x_k^T S_\sigma u_k + u_k^T R_\sigma u_k \quad (11.5)$$

$$Q_\sigma = \int_0^{T_s} A_\sigma^T(\tau) Q_c A_\sigma(\tau) d\tau \quad (11.6)$$

$$S_\sigma = \int_0^{T_s} A_\sigma^T(\tau) Q_c B_\sigma(\tau) d\tau \quad (11.7)$$

$$R_\sigma = \int_0^{T_s} B_\sigma^T(\tau) Q_c B_\sigma + R_c d\tau \quad (11.8)$$

with $T_s = t_k - t$, see [Xu, 2017]. Starting from a continuous-time design, it is straightforward to create the discrete LQ controller. A structural difference appears in the discrete controller in the cross-coupling term S_σ , which is often dropped in MPC. Therefore, it is typically not possible to calculate the continuous cost from the discrete cost, and consequently derive a correct re-sampled counterpart. With access to the continuous design, the approach is to use the cost functions obtained from the discretized infinite-horizon LQ controller in the finite-cost MPC controllers, and include the cross-coupling term. This makes the controllers as similar as possible. In the evaluations, Matlab's `quadprog` is used to implement an MPC which includes S_σ .

The two MPC controllers have the same horizon expressed in the number of time units. As a consequence of this, the horizon expressed in number of samples is half as large for the slow MPC as for the fast MPC, i.e., $N_2 = N_1/2$. This means that in addition to having twice as long time available for solving

the MPC optimization, the optimization problem to solve is also half as large in the number of decision variables. This leads to the MPC formulation

$$\underset{\mathbf{u}}{\text{minimize}} \quad \sum_{n=0}^{N_\sigma-1} z_n^T \mathcal{Q}_\sigma z_n + \mathcal{V}_{f,\sigma}(x_{N_\sigma}) \quad (11.9a)$$

$$\text{subject to} \quad x((k+\sigma)T_s) = A_\sigma x(kT_s) + B_\sigma u(kT_s), \quad (11.9b)$$

$$Gz_n \leq g, \quad (11.9c)$$

$$\mathcal{Q}_\sigma = \begin{bmatrix} Q_\sigma & S_\sigma \\ S_\sigma^T & R_\sigma \end{bmatrix}, \quad (11.9d)$$

where $z = [x^T \quad u^T]^T$ is the augmented state vector, σ indexes the two models, $\sigma \in [1, 2]$, and T_s is the base sampling period. The equality constraint (11.9b) is obtained from Equation (11.3). The inequality constraints (11.9c) are the same for both controllers. The terminal cost $\mathcal{V}_{f,\sigma}(x)$ is individually obtained from the discrete Riccati equation, i.e.,

$$P_\sigma = Q_\sigma + A_\sigma^T P_\sigma A_\sigma - (A_\sigma^T P_\sigma B_\sigma + S_\sigma)(R_\sigma + B_\sigma^T P_\sigma B_\sigma)^{-1}(B_\sigma^T P_\sigma A_\sigma + S_\sigma^T). \quad (11.10)$$

The controller uses a terminal cost $\mathcal{V}_{f,\sigma}(x) = x^T P_\sigma x$, but does not define a terminal set. Formally, the stability of the controller depends on the following assumption

ASSUMPTION 11.1

For any feasible state $x \in \mathcal{X}$, the horizon N_σ of mode σ is large enough to ensure that

$$x_{N_\sigma} \in \mathcal{X}_f, \quad \text{where} \quad \mathcal{X}_f \subseteq \mathcal{X},$$

and

$$x_{k+1} = (A + BK_\sigma)x_k \in \mathcal{X}_f \quad \forall x_k \in \mathcal{X}_f. \quad \square$$

That is, the final state x_{N_σ} will always be in a region \mathcal{X}_f around the origin, where the stabilizing control law $\kappa_f = K_\sigma x_k$ is invariant.

Device execution

Three alternatives are considered in terms of when the local MPC should execute. Figure 11.2 illustrates these alternatives in a timing diagram. The dashed activities for MPC₁ are executions when the response time is larger than h , and hence, feedback from the cloud MPC is not available. For MPC₂ the dashed activities show wasted executions, which have been invalidated by

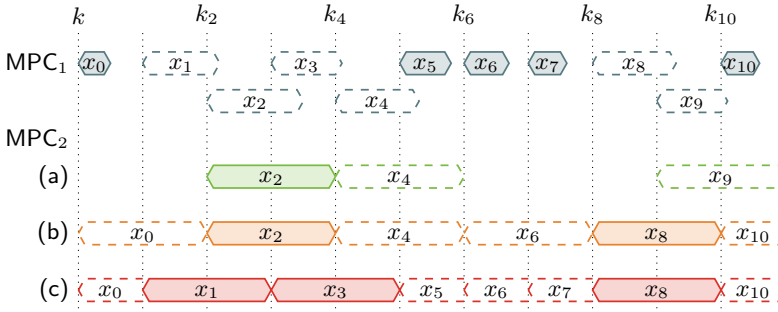


Figure 11.2 Three execution alternatives for the local device MPC. In (a), the client MPC does not start as long as there are responses available from the network. In (b), the client MPC starts execution periodically. In (c), the client MPC is restarted when input arrives from the remote controller. The text in each execution shows associated sampled state. The dashed blocks show optimizations that are not used, i.e. wasteful executions.

responses from MPC₁. The text inside the activities represents the sampled state used in the optimization.

In Alternative (a), the execution of the local MPC starts after the cloud MPC has failed to respond within the deadline. In Alternative (b), the local MPC runs periodically but the results are disregarded as long the cloud MPC returns a reply. In Alternative (c), the local MPC also runs periodically, but it is interrupted and restarted immediately if a response arrives from the network. The high frequency MPC₁ is invoked in every time step kT_s , also if it did not return any results in the last invocation. The two rows for MPC₁ illustrate that parallel MPCs can be started in the cloud, even though previous requests are still being processed. Being able to start new instances without concern of overloading the system is a convenient consequence of using a cloud platform. This provides the opportunity to periodically start new optimizations, while also collecting delayed responses. Here, however, all late responses are discarded, and cannot be used for control, even if they arrive only slightly delayed.

Control signal selection

Now consider the control signal selection mechanism. Denote the latest sequence of control inputs from MPC₁ as $u_{\sigma=1}^*$, and the latest sequence from MPC₂ as $u_{\sigma=2}^*$. The output of κ is the control signal that is applied to the plant. κ executes at the base frequency $1/T_s$, and provides the switching mechanism through the four cases in Table 11.1. Cases A1 and A2 apply the closed loop constrained control of the two MPCs. When the cloud MPC

Table 11.1 Four case selections in κ , in order of priority.

Case	κ	Description
A1	$u_{\sigma=1}^*(0)$	Applies MPC ₁
A2	$u_{\sigma=2}^*(0)$	Applies MPC ₂
A3	$\zeta(u_{\sigma=1}^*, x_k)$	Chooses an action during switching
A4	$K_d x$	Use LQR when no MPC is available

responds within the sampling period, T_s , A1 is available, and the output of MPC₁ is applied. When the cloud MPC fails to respond, the possibility of applying A2 depends on the device execution alternatives in Section 11.2. For instance, in Figure 11.2, when MPC₁ does not respond for state x_1 at time k_2 , Alternative (c) can apply A2 at time k_3 (using state x_1), but the Alternatives (a) and (b) cannot apply A2 until at time k_4 (using state x_2).

A3 implements a function used in the intermediate stage when the cloud MPC has missed its deadline and the system is waiting for A2 to become available, i.e. waiting for the device MPC to finish. In Figure 11.2, at time k_2 , all three alternatives must use A3. MPC₁ successfully returns a result for state x_0 , but then fails to respond to state x_1 in the following step. At this point, there is no local MPC₂ result available. At time k_3 , execution Alternative (c) is ready to apply A2, while Alternatives (a) and (b) must apply A3 again.

Due to the half rate of MPC₂, after at most two intermediate samples for Alternatives (a) and (b), and one sample for Alternative (c), the device MPC should provide a result. If it does not, the final fallback mode, A4, is used. This final action uses unconstrained LQ control to stabilize the system in case both MPCs fail. For Alternative (c) in Figure 11.2, A4 would be applied at time k_3 , had the local MPC failed. Similarly for Alternatives (a) and (b) this can happen at time k_4 .

One limitation that is not considered, is the possibility that the solver could detect an infeasible problem and abort before its deadline of $2T_s$ has passed. If this had been the case, A4 could have been applied earlier. A second limitation is that the control signal from MPC₂ is discarded if A1 is applied while waiting for MPC₂ to finish. This happens at time k_2 for Alternative (b) in Figure 11.2. MPC₂ has had time to evaluate state x_0 when MPC₁ fails to deliver a result for state x_1 . x_0 was already applied through a response from MPC₁ in the previous step, and the result from MPC₂ is therefore invalid. An alternate choice is to view the previous input from MPC₁ as a disturbance, not an overriding signal, and apply MPC₂ for state x_0 .

With the modes in place, it remains to define the function $\zeta(u_{\sigma=1}, x_k)$ for the switching sequence, A3. Four versions of this case action are listed in Table 11.2. The first, A3.1, applies the unconstrained LQ controller during

Table 11.2 Four choices of action A3 in Table 11.1. k_r is the latest response time of MPC₁, when $u_{\sigma=1}$ was received.

Case	κ	Description
A3.1	$K_d x_k$	Use LQ control
A3.2	$u_{\sigma=1}^*(0)$	Hold the constrained control signal
A3.3	$u_{\sigma=1}^*(k - k_r)$	Apply open loop control
A3.4	$u_{\sigma=1}^*(0) \vee u_{\sigma=1}^*(1)$	Conditional one step open loop

switching. The second, A3.2, simply holds the previous control signal. The third, A3.3, uses the pre-calculated control signal sequence from the last invocation of MPC₁, i.e., runs MPC₁ in open loop, while waiting for MPC₂ to complete. In the fourth mode, A3.4, the control signal is held during the execution of the local MPC. With this option, κ can choose to apply one step of open loop, and then hold the control signal. The choice depends on which Alternatives (a)-(c) that is used. For Alternative (a), κ applies $u_{\sigma=1}(1)$. For Alternative (c), it applies $u_{\sigma=1}(0)$. For Alternative (b), it applies $u_{\sigma=1}(0)$ if the local MPC started in the time slot when the latest network response was received, otherwise it applies $u_{\sigma=1}(1)$. This ensures that, in the nominal case, the state prediction used by the local MPC is correct. More advanced control laws than the ones in Table 11.2 are possible, e.g., one could use soft constraints, robust MPC, or decaying strategies such as the one in Chapter 10.

11.3 Evaluation

The simulations are implemented in Simulink, using a continuous time plant model and TrueTime [Cervin et al., 2003; Department of Automatic Control, Lund University, 2019] to implement the switching controller. Matlab's quadprog solver is used for the MPC implementation. Simulations are used in order to be able to easily compare the various controllers' nominal behavior and how they behave during switching.

Constraints in the evaluations are set to to $x_{ub} = x_{lb} = [0.55 \ 1 \ 0.7854]$ and $u_{ub} = -u_{ul} = 10$. The continuous-time costs are $Q_c = \text{diag}([1500 \ 20 \ 0])$ and $R_c = 1$, chosen empirically. The rates 30 Hz and 15 Hz are used in the evaluation with controller horizons 10 for MPC₂ and 20 for MPC₁. The setpoint for the recovery LQR is not updated while MPC₂ is producing a result.

Performance metrics

Three metrics are used for performance evaluation: the integrated continuous-time cost, the integrated constraint violation, and the maximum constraint

violation. The cost metric

$$\mathcal{C}(t) = \int_0^t (x(s)^T Q_c x(s) + u(s)^T R_c u(s)) ds. \quad (11.11)$$

provides a comparison to the cost using the shared performance objective. The constraint violation metrics measure the robustness of the approach. Preferably, there should be no constraint violations, but since the system can enter sequences of both open loop control and unconstrained LQ control, this is unavoidable. The integrated constraint violation and maximum violation metrics are defined as

$$\mathcal{V}(t) = \int_0^t v(s) ds \quad (11.12)$$

and

$$\mathcal{V}_{max}(t) = \max(v(s)), \quad s = 0, \dots, t, \quad (11.13)$$

where the violation $v(t)$ is

$$v(t) = \max(0, |x(t)| - |x_{ub}|) + \max(0, |x(t)| - |x_{lb}|). \quad (11.14)$$

The max operation is applied element wise, and $v(t)$, $\mathcal{V}(t)$, and $\mathcal{V}_{max}(t)$ are, thus, vector valued. Only the first value, the controlled state x_1 , is used in the evaluation.

Network and plant disturbances

In the evaluations, a random disturbance enters the system as w , affecting the input to the plant such that

$$x((k+1)T_s) = A_1 x(kT_s) + B_1(\kappa(\cdot) + w(kT_s)), \quad (11.15)$$

i.e., the disturbance adds a random offset to the input.

The simulated scenarios are chosen to, in particular, evaluate the performance during switching between the controllers. This is done by using a fifty-fifty chance of packet loss, or in terms of network request latency, τ , $P(\tau < T_s) = 0.5$. In practice, it is expected that under ordinary conditions, control is performed by the cloud MPC, except for occasional deadline overruns. Similarly, if the network is down, the control will be done by the client MPC. For comparison, results are provided also for the uninterrupted cloud MPC and client MPC.

Synthesis validation

Before studying the switching design, the individual controllers are evaluated to see that they work according to expectations. Figure 11.3 shows the step

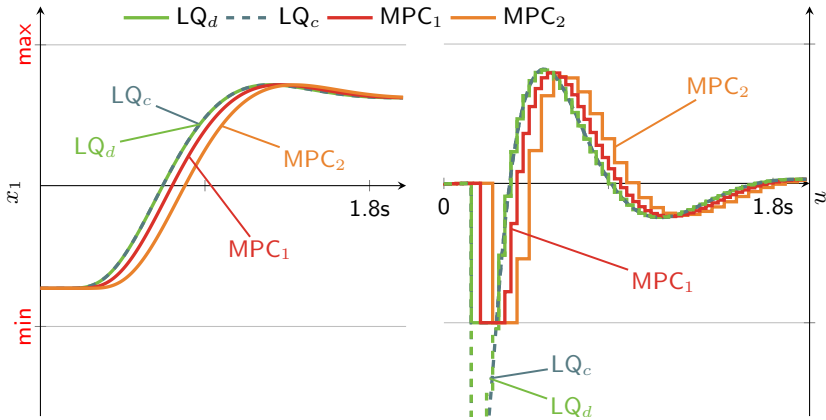


Figure 11.3 Step response of LQ_c and derived controllers. The MPCs apply the control signal after a one sample delay. Left: Evolution of state x_1 (the ball position). Right: The applied control signal.

responses of four nominal closed loop systems, in terms of state x_1 and the control signal u . The controllers are designed according to the procedure in Section 11.2. As seen in the right plot of Figure 11.3, the two unconstrained LQ controllers violate the input constraint, and u is therefore saturated. This is illustrated by the dashed lines showing the control signals requested by LQ_c and LQ_d .

The response of the discrete-time LQR is very close to the continuous-time counterpart, which is seen from LQ_c overlapping LQ_d in the left plot. The same does not apply to MPC_1 , due to its built-in delay of one sampling period, and the difference is even more prominent for MPC_2 , since the sampling period is twice as long. For the cost metric given by Equation (11.11), the excess cost of LQ_d compared to LQ_c is only 0.5%. Due to the delays, the two MPCs have additional costs of 11% for MPC_1 , and 26% for MPC_2 . Introducing the remote controller, MPC_1 , can reduce the excess cost from MPC_2 by up to 15 percentage points, in this scenario.

Switching alternative

To see how the switching works and study its effect on performance, an experiment consisting of repeated step changes is used in combination with the network disturbance. A snapshot from this experiment is shown in Figure 11.4. The figure shows the response for six different controllers. LQ_d provides a reference, unconstrained controller. MPC_1 shows the result with reliable connectivity to the cloud, and MPC_2 the result when the cloud is disconnected. The remaining controllers implement the execution Alternatives (a)-(c) from Section 11.2.

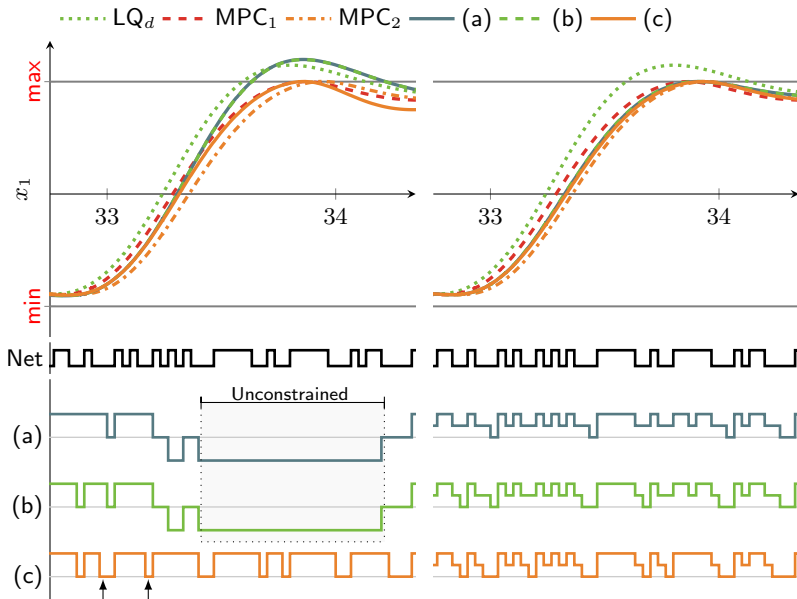


Figure 11.4 Example of switching control during a step change. Left: Holding the latest control input when switching. Right: Applying open loop control when switching.

The upper plots show the trajectory of x_1 , with min and max showing the constraints on x_1 . Below this trajectory, there is a timing plot labeled *Net*, which shows the network response. When this signal is high, timely responses are received from the cloud. When this signal goes low, it indicates that MPC_1 (for cases A, B, and C, in the figure) cannot act due to long delays in the network. Below the network plot are timing diagrams showing the current source of control input for the switching controllers. The plots are centered around a gray line. On the gray line, the device MPC is used, i.e. MPC_2 . When the plot goes above the gray line, the system is using control input from MPC_1 . This can be in closed loop or, when MPC_1 fails to respond, a held signal or the use of a previously calculated open loop sequence. When the plot is below the gray line, constrained control has failed and the system uses LQ_d .

On the left side of Figure 11.4, the strategy A3.2 from Table 11.2 is used, i.e. when switching, $\kappa(\cdot)$ holds the previous output. This choice is risky, especially for Alternatives (a) and (b). The absence of control causes MPC infeasibility in both MPC_1 and MPC_2 during 33.3s to 34.2s. This results in a sequence of invocations of the unconstrained LQ controller, and, conse-

quently, constraint violations. Alternative (c) does not violate the constraint, since it is able to apply local MPC control faster when the network fails. This is observed by looking at the sequence of events leading up to the unconstrained LQ control. On two occasions when Net becomes low, mode (c) more quickly changes to local MPC control, compared to (a) and (b). These occasions are marked by two arrows under the plot for (c). Since the network is unresponsive, (a) and (b) hold the control signal constant, while (c) is able to apply the local constrained control. The faster response of (c) allows it to remain on a feasible path, and it does not have to apply unconstrained control. Note that no disturbance is added in this scenario, except for the disturbance caused by not being able to apply closed loop control in the switching sequence.

On the right side of Figure 11.4, the simulation is rerun using the switching strategy A3.3 from Table 11.2, i.e. applying open loop control. Alternative (a)-(c) all maintain MPC control. In the timing plots, there are now smaller steps from the maximum level to the central, gray line. This shows when the system runs in open loop. The number of open loop inputs and use of the client MPC differ between the alternatives, but there are no drops below the gray line, i.e., no unconstrained control is necessary. Being a nominal simulation, this simply indicates that the system does not deteriorate because of the switching, and that the implementation works.

Performance

The performance of the system must be considered in terms of both the cost and the amount of constraint violations, as defined in Section 11.3. This section looks at longer simulations, and shows the final values of the three defined metrics. The experiments execute 360s for a total of 10.8k samples. In the step response simulations, the step looks as in Figure 11.4, and the setpoint is held for two seconds, causing a total of 180 step changes. In the disturbance rejection simulations, the setpoint is constant at 0.54, close to the x_1 constraint of 0.55. The disturbance w in Equation (11.15) is drawn from the standard normal distribution, i.e. $\mathcal{N}(0, 1)$, with the same sequence used in all examples.

Figures 11.5 and 11.6 shows the results of the step response simulations. All alternatives are better than the device MPC, with the lowest costs occurring when using A3.1. This case uses unconstrained LQ control during switching, which gives a low cost in Figure 11.5, but is also the reason for the constraint violations in Figure 11.6. Compared to unconstrained LQ control though, the total violations are small. Being a nominal scenario, MPC₁ and MPC₂ do not violate constraints, and neither does open loop control (A3.3).

In the simulations shown in Figures 11.7 and 11.8, the setpoint is constant, but the disturbance w is active. The system can be forced into sit-

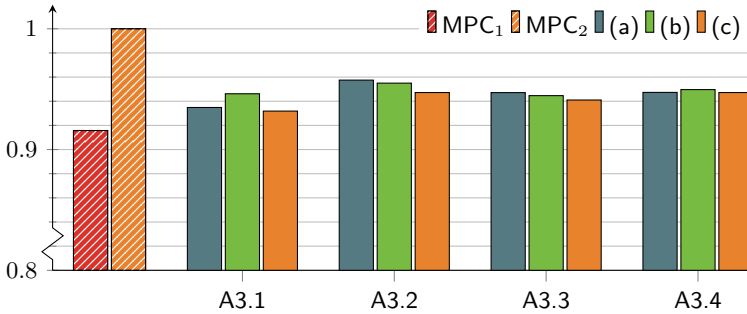


Figure 11.5 Cost of 180 step changes relative to MPC₂.

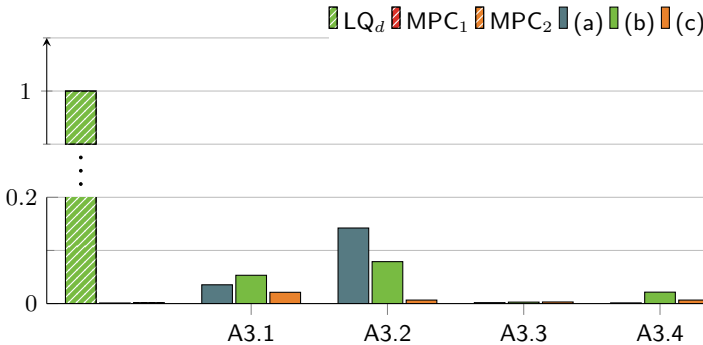


Figure 11.6 Constraint violation metric relative to LQ_d, after 180 step changes.

uations where the constraints will be violated, also when controlled by the individual MPCs. When this happens, the MPC problem becomes infeasible and LQ_d is applied for recovery, also in the MPC₁ and MPC₂ simulations. The results in Figures 11.7 and 11.8 show three things. First, all switched controllers outperform the local device MPC. This applies to both cost and constraint violations. Second, none of the switched alternatives fare worse than the unconstrained LQ control in terms of the constraint violations in Figure 11.8. Third, while using unconstrained LQ control as the recovery mechanism can cause some violations in the nominal case (see Figure 11.6), its fast response is useful in the disturbance scenario. The effect is seen in Figure 11.8, where mode (c), although fast to apply local constrained MPC control, consistently has the worst constraint violation. Hence, the faster, unconstrained LQ control is preferred over the constrained local MPC to handle the disturbance.

In summary we see that, with a fifty percent chance of losing the cloud

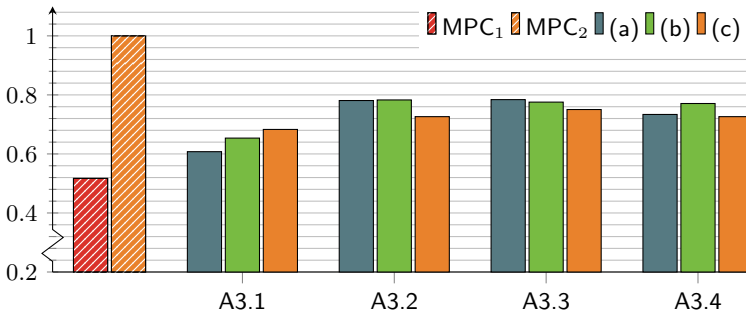


Figure 11.7 Cost, relative to MPC₂, after a sequence of 10.8k random input disturbances.

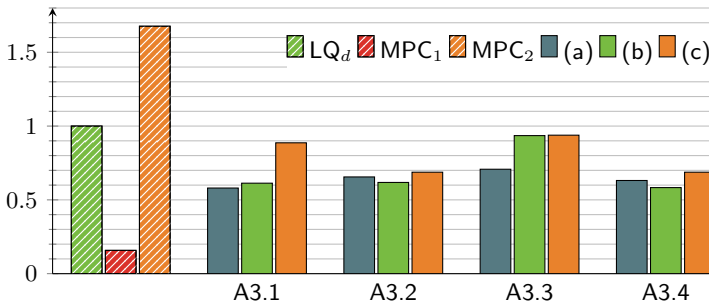


Figure 11.8 Constraint violations metric, relative to LQ_d, after a sequence of 10.8k random input disturbances.

MPC response, all cases perform well in the nominal case. The choice of switching mechanism A3.1 provides the best performance in terms of cost, but can cause constraint violations, as a consequence of the unconstrained control during the switch. As expected, A3.2, which does not apply any control during the switch, has the worst score in terms of both cost and constraint violations. The better nominal performance of Alternative (c), compared to Alternatives (a) and (b), is attributed to a faster response from the device MPC when the cloud MPC fails. The two MPCs are derived to be very similar, and so the momentary use of the device MPC causes only a small disturbance for the high frequency control. In the disturbance scenario though, Alternative (c) does not outperform the others. Its frequent use of the device MPC is now counterproductive, since the delay of the controller is problematic in the presence of the disturbance. To reject the disturbance, A3.1 performs well in the combination of cost and constraint violations. When the MPC control fails, this case has the benefit of a non-delayed optimal LQ controller to quickly counteract the disturbance. Importantly though,

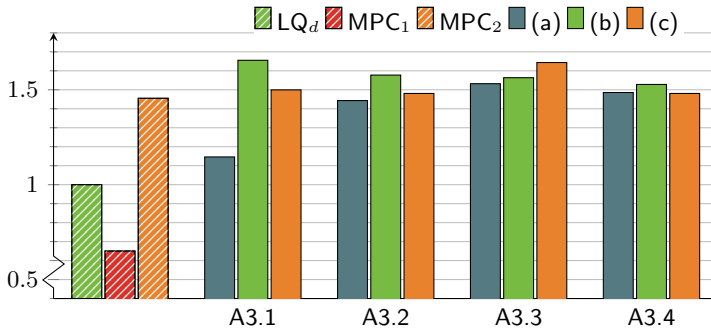


Figure 11.9 Maximum constraint violation over the course of the disturbance simulations.

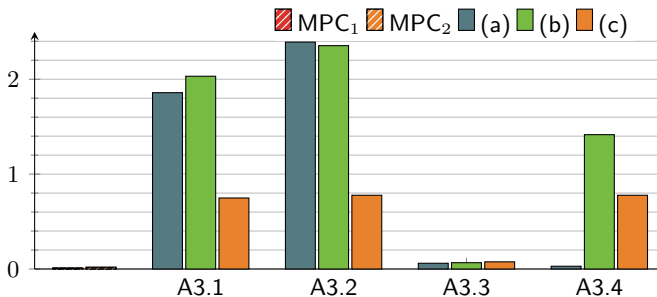


Figure 11.10 Maximum constraint violation over the course of the step change simulations. These are shown relative to the constant overshoot of LQ_d.

all cases performs better than the standalone local MPC in terms of both cost and constraint violations, in the nominal case and in the disturbance scenario.

A downside of the approach is illustrated in Figure 11.9. The bars in this figure show the maximum constraint violation of Equation (11.13) relative to LQ_d, over the course of the disturbance experiment. It is seen here that all modes have a worst case constraint violation that is larger than that of the unconstrained LQ_d in the disturbance simulation. Because of its lower sampling rate, this is also the case for the low frequency device MPC. Several of the modes however, also have a somewhat larger value than MPC₂. Mode (a) provides the best result as a consequence of frequently using the LQR.

The results are completed by Figure 11.10. The figure shows the maximum constraint violations in the step simulations. This figure should be approached with care. The violations happen on single occasions, at different

times during the execution, and depending on the dynamics caused by failed requests and frequently changing the setpoint. One thing to note is that the open loop results in A3.3 are small but non-zero. This is due to differences between the responses from the low and high frequency controllers. While it is so small that it is barely visible, there is also a small error for MPC₂ due to infeasible optimizations near constraints.

Alternative (a) in A3.4 has a low value and this mode also avoids prediction error. Alternative (c) holds the control signal in both A3.2 and A3.4. Alternative (b), which shows the largest values, alternates between the other two sequences depending on request loss timing. In A3.2, (a) and (b) are worse than (c) because they hold the control signal more often. Finally, A3.1 shows that the quick application of constrained control in alternative (c) is good for the maximum violation in this nominal scenario. The values when using the LQR would be lower if there was more time for the state to settle between setpoint changes. One way to improve this result is to hold the setpoint in the LQR recovery, but this will also increase the cost in Figure 11.7.

11.4 Conclusions

This chapter has proposed and demonstrated a two-tiered MPC architecture. The two MPC controllers were designed to be as equivalent as possible, except for the different sampling rate. This strong similarity was aimed at limiting the detrimental effects of arbitrarily switching between the two controllers. The lower rate MPC cannot always provide a result, so a third alternative must be applied at times. The MPCs are implemented using hard constraints, and can become infeasible. Alternatives for applying a control signal in these situations were considered. With a twofold reduction in sampling frequency on the client, the technical considerations for the execution of the client MPC were detailed and switching alternatives were evaluated.

As expected, all combinations improved on the performance of the local device in terms of accumulated cost and error. Possible concerns are around maximum constraint violations which could increase in relation to the constrained local controller. From available results, mode (c) in combination with A3.4 has the positive aspect of consistently showing low constraint violations. This is the mode that synchronizes the start of the local MPC with the remote, and which uses the conditional one step open loop for the intermediate control action. Mode (a) in combination with the LQ recovery, A3.1, has the benefit of the client MPC being mostly idle, while providing low cost and good disturbance rejection. A downside is that the LQ might accelerate the system to become infeasible. Mode (b) represents switching between two independently executing controllers. This mode does not excel in any combination.

12

Explicit Recovery

So far, the offloaded optimization has executed an ordinary MPC problem, with recovery implemented only on the client. Stability has been asserted through experiments and constraint satisfaction has not been required in the transition from remotely assisted to local control. In this chapter, a recovery action is evaluated in the cloud, by way of introducing it into the MPC. This is aimed at providing a recovery mode in which the nominally system is stable and can ensure that system constraints are not violated, also in the event that the remote controller is suddenly disabled. This is implemented on the reference platform, but extends directly to more complex systems because of the explicit, on-line evaluation.

Different from Chapters 10 and 11, the goals of the remote do not have to coincide with the basic client controller. The reason for having a remote with a different objective can be context dependent. For instance, an autonomous transport might be allowed more overshoot in its response when the position and actions of its peers are known. If disconnected from the remote systems, the actions, for safety reasons, could be very different. Another reason could be the model of the plant. The device controller may be based on a local linear model, while the cloud evaluates a non-linear version. The more detailed model could allow for a different objective.

12.1 Setup

As usual, the setup starts from two independent controllers, the basic, local controller, and the more advanced controller, executed in the cloud. In this chapter, the basic controller is referred to as the *device* controller. The responsibility of the *client* is as usual to initiate optimizations in the cloud and apply the subsequent control actions. Different from previously is that the graceful degradation (or recovery mode), κ_2 in Figure 9.1, is provided by the

This chapter is based on [Skarin and Årzén, 2021]

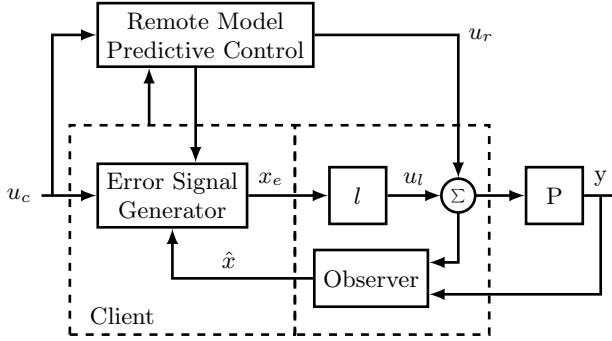


Figure 12.1 Supporting, remote controller structure

result from the cloud. To ensure constraints, the control problem executed in the cloud must evaluate the response of the recovery mode κ_2 .

The following examination continues to exercise the reference system, assuming full state knowledge and a linear device controller. The setpoint governor from Chapter 10 is used to limit the error entering the system, so that the device controller can handle setpoint changes without violating constraints. The reference system can then be modeled as illustrated in Figure 12.1. In the figure, u_c provides the command signal to the system. The error signal generator provides the governing function, translating the command into a state error, x_e , which is passed to the device controller. To do so, it obtains a state estimate, \hat{x} , from an observer and information from the remote controller. The remote can also provide a feed-forward signal, u_r . The arrow leading from the client to the remote controller initiates new calculations in the cloud, and the arrow leading back provides results from MPC optimizations.

The configuration in Figure 12.1 can be likened to the state-space version of a 2-Degrees-of-Freedom (2-DoF) structure, presented in [Åström and Wittenmark, 2011], with the model and feed-forward generator replaced by an active part; the MPC and the error signal generator. The original 2-DoF translates u_c into a desired state x_m and an open loop control signal u_{ff} , then applies

$$u(k) = l(x_m(k) - \hat{x}(k)) + u_{ff}(k), \quad (12.1)$$

where l is the device controller in Figure 12.1. This is straight forward to recreate, using the control signal and state predictions from the MPC. However, we are interested in a remote that can take full control, i.e., also handle disturbances, and a client that is capable of tracking the reference with or without the remote assistance. Therefore, u_c must be passed to the local controller, allowing it to also act on the command.

The typical, basic assumptions are used:

- A-1 The remote and the client are synchronized in sampling and actuation time,
- A-2 The control input has a deadline of one sampling interval, from sampling to actuation, late arrivals are discarded and equal to packet loss,
- A-3 The actions of the remote controller are always preferred, and
- A-4 The remote controller is not necessary in order to perform tracking and stabilize the system.

Assumption A-1 is easy to achieve with an offloading controller. Assumption A-2 follows how the offloading control has been implemented so far. Assumption A-3 is natural as long as the remote provides better performance, but the requirement makes the client behavior predictable. The final assumption, Assumption A-4, states that the remote predictive controller can be removed from Figure 12.1. This implies that there is some state to which the client can return the plant, and the configuration of the error signal generator, then disable the remote controller function. The assumption does not imply that the remote can be arbitrarily removed after it has started to apply its control action.

12.2 Controller

Preliminaries

Assuming perfect state information, the client control law is given by

$$u_l(k) = l(x_e(k)), \quad (12.2)$$

where x_e is the state error generated by the error signal generator, and $l : R^n \rightarrow R^m$ is a controller, translating state error to a control signal vector. The device controller could take any form, but in accordance with Assumption A-4, it is asymptotically stable for the set $\mathcal{L} \subset \mathbb{R}^n$ of admissible initial states. If hard constraints are defined on the system, \mathcal{L} must be in the interior of the constraint set and positively invariant (Definition 2 in Section 4.2) under the client control law (12.2).

To ensure the positively invariant property, the state error is limited by the error signal generator through a function ψ_1 ,

$$x_e = \psi_1(u_c, \hat{x}) \quad (12.3)$$

This function is active when the client is acting stand-alone¹. The admissible set of the client, through the use of (12.3), is referred to as \mathcal{L}^+ . This set is

¹This chapter concentrate on tracking a reference signal and does not consider the details of how ψ_1 impacts the regulator problem.

assumed to be positively invariant under (12.3) and (12.2). Finally, $\mathcal{L}^+ \subseteq X_c$, for some system constraints X_c and all sets are closed, convex, and contain the origin in their interior.

The remote controller is based on the optimal control problem

$$\underset{u_r^{0,N}, \kappa^{0,N}}{\text{minimize}} \quad \sum_{k=0}^N J(z_k, u_r^k, \kappa^k) \quad (12.4a)$$

$$\text{subject to} \quad z_{k+1} = f_r(z_k, u_r^k, \kappa^k), \quad (12.4b)$$

$$g(z_k) \leq 0, \quad (12.4c)$$

$$h(u_r^k, \kappa^k, z_k) \leq 0, \quad (12.4d)$$

$$z_0 = \hat{x}(t), \quad (12.4e)$$

$$x_N \in \mathcal{T}_f, \quad (12.4f)$$

where \hat{x} is the observed plant state, J is the cost function, κ^k is a local control law, g and h implement control and input constraints, and \mathcal{T}_f is the terminal set. The model used in f_r can be different from the one used in the local client (f) but is assumed to be at least as precise. State and input constraints are freely defined, independent of the assumptions and requirements of the client controller. The terminal set, \mathcal{T}_f , is assumed to bring the plant state into the admissible set of the client controller. It is straight forward to assume no knowledge about the error signal generator and design the MPC with the standard requirement that $\mathcal{T}_f \subseteq \mathcal{L}$.

Returning to Equation (12.3) and Equation (12.1), if κ^k is l , and ψ_1 is realized as

$$\psi_1(u_c, z_k, \hat{x}_k) = \hat{x}_k - z_k, \quad (12.5)$$

i.e., the difference between the predicted and observed state, then, using the nominal prediction and control signal, the 2-DoF structure is obtained. This can be used to implement the action of a robust tube-MPC. A tube-MPC uses an ancillary control law to keep the actual state \hat{x}_k close to the nominal z_k . It accounts for the disturbance rejection of $l(x_e)$ by tightening the state and input constraints in the optimization. One might consider replacing l online, and supporting an M-step sequence $u_r^{0,M}, x_m^{0,M}$, where a maximum disturbance is assumed in each step. Such a strategy can be used to robustly handle delays and packet losses but requires a reliable combination of disturbance model and compensating controller. It also requires availability of predictions and control signals over the horizon. We consider now an alternative that does not attempt to follow the predicted nominal path if the MPC fails but instead reverts to use the device controller. There must be a recovery option when the remote goes off-line and simply switching back to (12.2) in combination with (12.3) is not a safe alternative.

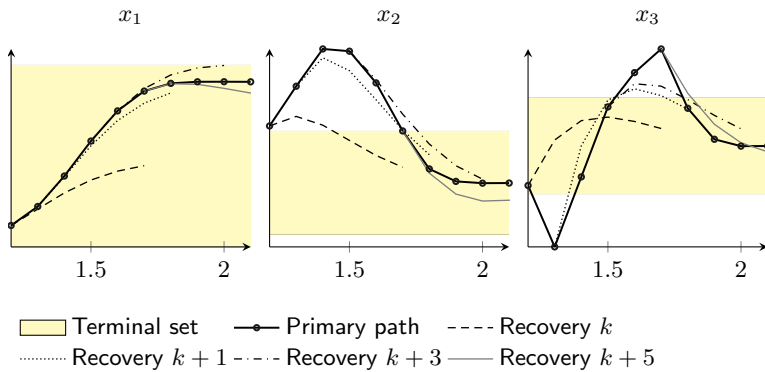


Figure 12.2 Predicted trajectory and four recovery paths. x_1 , x_2 and x_3 are plant state variables. Yellow areas show the recovery terminal set. Some of the trajectories extend beyond the figure on the horizontal time axis.

Control strategy

The aim is to derive a safe control strategy from the structure in Figure 12.1, which allows for the remote system to come and go. The remote is limited to making temporary changes in the error signal generator, keeping the device configuration unchanged. This is not a necessary limitation, it is straight forward to replace the device controller l if the situation allows. A reason for manipulating only the error signal generator is that this separation is straight forward with the linear device and client controller. This setup, with linear local control, does not limit the definition of the assisted controller but makes the recovery control sequence easy to implement and fast to evaluate.

To explicitly ensure that the system is able to recover, (12.4) is extended with a set of constraints that define *recovery paths*. An example of such recovery paths is shown in Figure 12.2. The figure shows the result of one optimization, with $N = 10$, and four out of ten recovery paths (one for each step in the horizon). Solid black lines show the primary path for each system state, x_1 , x_2 , and x_3 . The included recovery paths are at offset zero, two, three and five, on the prediction horizon. The figure shows how, with each step on the solid black line, the controller has found a recovery path that ensures that the plant state is within the device constraints (yellow) within five time steps. This creates a nominal guarantee that the constraints in the remote controller (not shown in this figure) will not be violated in the event of failure. The objective of the recovery is only to keep the system operational. It is of essence that the inclusion of a recovery mode does not substantially degrade remote control in closed loop, but low performance during recovery, in terms of the objective defined for the remote control mode, is acceptable.

Controller extension For every step in the prediction horizon of Equation (12.4), there must be a recovery path that 1) stays within constraints, and 2) ends in a safe terminal set. The controller must evaluate N recovery paths. The paths are denoted w_j^k , where k relates to the state z_k in (12.4) and j indexes steps in the recovery path. The additional set of constraints

$$\begin{aligned} w_{j+1}^k &= f_r(w_j^k, \kappa_2(w_j^k)), \\ w_0^k &= z_k, \quad w_{N_r}^k \in \mathcal{T}_r, \end{aligned} \quad (12.6)$$

where $j = k, \dots, k + N_r$, defines recovery paths, using the recovery mode κ_2 . The recovery horizon, N_r , sets the number of time steps within which a recovery path must end in an acceptable state, as defined by the new terminal set \mathcal{T}_r . In addition, the states w_j^k must also satisfy the MPC constraints. Inserting into Equation (12.4) and fixating the local controller to κ_3 , the remote control problem becomes

$$\underset{u_r^{0,N}}{\text{minimize}} \quad \sum_{k=0}^N J(z_k, u_r^k, \kappa_3(z_k)) \quad (12.7a)$$

$$\text{subject to} \quad z_{k+1} = f_r(z_k, u_r^k, \kappa_3(z_k)), \quad (12.7b)$$

$$w_{j+1}^k = f_r(w_j^k, \kappa_2(w_j^k)), \quad k \geq 1 \quad (12.7c)$$

$$g(z_k) \leq 0, \quad h(u_r^k, \kappa_3(z_k)) \leq 0, \quad (12.7d)$$

$$g(w_j^k) \leq 0, \quad h(\kappa_2(w_j^k)) \leq 0, \quad (12.7e)$$

$$z_0 = \hat{x}(t), \quad w_0^k = z_k, \quad (12.7f)$$

$$x_N \in \mathcal{T}_f, \quad w_{N_r}^k \in \mathcal{T}_r \quad (12.7g)$$

$$j = 0, \dots, N_r. \quad (12.7h)$$

Device recovery First, consider inserting the device controller into Equation (12.7c) to provide

$$w_{j+1}^k = f_r(w_j^k, l(\gamma_j^k w_j^k)), \quad (12.8)$$

where paths are traversed using the device controller, l , but predicted using the remote's model of the plant. A scaling γ_j^k is added, allowing that the client modifies the observed state during the recovery. This extends the governing function ψ_1 . Note that the original limiting function (12.3) is not taken into consideration in the remote and is not in effect during this sequence. Instead, $\psi_2(\gamma^{i,i+N_r}, x_e(k), k) = \gamma_{k-i}^i x_e(k)$, can adjust the client response, and since l is linear $l(\gamma_j^k x_e(k)) = \gamma_j^k l(x_e(k))$. If a solution exists for a set of gains γ_j^k , they translate into a sequence of setpoints that can be generated by the error signal generator.

Using a sequence of scalar γ_j^k , the recovery transition can scale the control action but the relative objectives of the controller do not change. These objectives will limit the potential of the recovery but also has other complications. The equality constraint for the recovery paths becomes

$$w_{j+1}^k = Aw_j^k + \gamma_j^k Bl(w_j^k) \quad (12.9)$$

and adding γ_j^k as optimization variables does not fit the linear and quadratic reference MPC. Also, either γ_j^k must be predefined and not change between optimizations, or γ_j^k must be included in the optimization cost function in (12.4). In practice, the state will also deviate from the nominal path, which can cause problems if the recovery ends outside the terminal set \mathcal{T}_f . The experiments will show examples where the device controller is used directly, to illustrate some limited gains when doing so, but γ_j^k is not optimized for. Instead, further specialization of the recovery mode is added.

Recovery mode To really be useful in recovery, the client must be able to counteract actions taken by the remote controller. This may not be the case when executing using the objective of the device controller. The consequence is that the admissible control of the remote controller will be limited. A specialization is created by replacing γ_j^k with a matrix, denoting it Γ . Two ways in which it can be ensured that the recovery provides a proper counter action are considered.

- R1) Use the matrix transformation Γ to define a recovery controller. Let the client control law be $u_l(k) = K\Gamma x_e(k)$, $x_e(k) \in \mathbb{R}^n$ and $\Gamma \in \mathbb{R}^{n \times n}$. A set of temporary objectives can be defined to generate a recovery control law K_f , then find Γ so that $K\Gamma = K_f$. Assuming that $K \in \mathbb{R}^n$ and contains no zeros, Γ is a diagonal matrix scaling each element of K . This fits within the defined framework of using the error signal generator as a means of implementing the recovery. It is optional to make \mathcal{T}_r in Equation (12.6) equal to the terminal set \mathcal{T}_f .
- R2) Define a new command signal \tilde{u}_c which tells the system where to go if the remote is lost. For instance, returning to the origin in \mathcal{L}^+ . Let χ be a function translating the command signal to a state setpoint. The terminal set \mathcal{T}_r in Equation (12.6) is defined as a region around $\chi(\tilde{u}_c)$. The recovery command \tilde{u}_c is used in place of u_c during the recovery procedure as the system approaches \mathcal{T}_r . Although a new setpoint is defined, the recovery sequence only has to reach \mathcal{T}_r .

Both of these choices can provide a necessary counteraction for recovery and, different from the issue with γ_j^k , are compatible with an ordinary linear system. In R1, if \mathcal{T}_r is different from \mathcal{T}_f , the recovery sequence can be short, relative to the number of steps necessary to reach the final setpoint defined by

u_c . Similarly, the action in R2 does not intend to reach the recovery setpoint, and depending on \mathcal{T}_r , only a few steps of recovery might be necessary. While R1 can provide important dampening in recovery, R2 can also ensure that, within a short time frame, the state moves sufficiently far away from some constraints. The two can be usefully combined.

Notes on implementation In an implementation, the remote must pass Γ and/or \tilde{u}_c to the error signal generator. Knowing \mathcal{T}_r is optional. Nominally, the recovery can run over the full horizon N_r to reach \mathcal{T}_r . If \mathcal{T}_r is known, the recovery sequence can also end early, or execute more than N_r steps to handle disturbances. K_f , \tilde{u}_c and \mathcal{T}_r are chosen freely. Notice also that all recovery states are contained in the constraints of the remote controller, even if \mathcal{T}_r or \mathcal{T}_f are differently defined. As such, it is also possible that the remote enforces some tighter constraint during its actions, including the recovery back to client mode, than what was initially considered in design of the client.

Properties of the strategy

This section provides some notes on extremes of the proposed strategy and stability considerations. First, consider what happens if the horizons are reduced to a minimum. Using $N_r = 0$, which is valid in Equation (12.6), creates an MPC controller limited by the domain of \mathcal{T}_r . This MPC verifies that all its predicted states are admissible by the client controller. This may intuitively not seem useful, but three things should be noted:

- 1) the restricting domain, \mathcal{T}_r , may be different from the constraints considered when designing the client,
- 2) the remote can work with an improved plant and disturbance model, and
- 3) the remote can implement a different control strategy.

Equation (12.4) can be reduced to a minimum by setting $N = 0$ and removing the terminal set \mathcal{T}_f . The remote then verifies that a state is admissible for the defined recovery controller, making the recovery controller an explicitly verified alternative to the local controller. With $N = 1$, and an excluded terminal set \mathcal{T}_f , the remote resembles an implementation of an MPC with a control horizon of one. Note that the closed loop response of the client should be included in the MPC terminal cost and the terminal state is verified through \mathcal{T}_r .

Now consider stability implications. The following is assumed:

- 1) the MPC defined by (12.4) is stable in closed loop,
- 2) the terminal set \mathcal{T}_f is an invariant set for the client controller (12.2), and

3) the client is invariant in \mathcal{L}^+ under (12.3).

As is customary, the nominal system and a perfect model of the plant (in the remote) is assumed. Let $z_{j|k-1}$ be the solution at time $k-1$ with j indexing the predicted time steps. When adding the constraints defined by (12.6), preserving the stability of the remote controller depends only on recursive feasibility, since no cost is added by (12.6) and the terminal conditions remain. This depends on $z_{j|k-1}$ remaining a valid solution. With $z_{j|k}$ the solution at time k , it must be ensured that $z_{k+N-1|k} = z_{k+N|k-1}$ is a solution. For this, there must exist a u such that $z_{k+N|k} = Az_{k+N|k-1} + Bu$ and where $z_{k+N|k}$ emits a valid recovery path w_j^{k+N} . The terminal conditions require that $z_{k+N|k}$ is in the terminal set \mathcal{T}_f , so we can state the stability criterion: the remote is guaranteed stable in closed loop, if all states in the terminal set, \mathcal{T}_f , admit a valid recovery path. With this condition, a state $z_{k+N|k}$ that is valid under (12.4) is also valid under (12.6).

Now assume that the requirement does not hold, i.e., the solution $z_{k+N|k} \in \mathcal{T}_f$ does not exist or \mathcal{T}_f is not defined. This results in a remote controller failure, forcing the system to enter the recovery mode. The recovery mode starts in z_k and follows a path that has been explicitly verified to reach the terminal set \mathcal{T}_r . It is required that client control is admissible and stable in \mathcal{T}_r , and we assume that the set is closed, so there can be no ambiguity. Then, traversing the full recovery path, i.e., applying $u_j = K\Gamma(\hat{x} - \chi(\hat{u}_c))$ for $j = k, \dots, k + N_r$, ensures $\hat{x} \in \mathcal{T}_r$. From there, the client will approach the original setpoint $\chi(u_c)$ and eventually the state enters \mathcal{T}_f . This provides a safe option.

If the remote control is lost but quickly recovers, it is preferable that the remote controller action can be applied. The just stated safe option does not allow this. To enable a switch back to the MPC it must be ensured that there is no *wind-up* of the cost. Denote the remote solution at time k as \mathbf{u}^* , and the remote cost of state x_k as $V_r(x_k)$. With closed loop remote control $V_r(Ax_k + Bu^*(0)) < V_r(x_k)$ through ensured stability. If failure occurs at time k , stability becomes an issue of the recovery control, $V_\delta(A\tilde{x}_k + K\Gamma\tilde{x}_k) < V_\delta(\tilde{x}_k)$, where \tilde{x} represents the translated state (see Section 12.3) and V_δ the cost defined in the recovery mode. Denote the state x_k after j steps of recovery control as $x_{k+j|\delta}$. The stability during recovery is assumed, as far as reaching the necessary states in \mathcal{T}_r , but it is not ensured that

$$V_r(x_{k+j|\delta}) < V_r(x_k), \quad (12.10)$$

i.e., the cost from the perspective of the remote controller is not guaranteed to decrease during recovery. The same thing applies when the client controller takes over, until reaching \mathcal{T}_f . While it may not be known beforehand if (12.10) holds, it can be evaluated online. This provides a practical, online solution, as has also been used in performance enhancing switched MPC [Magni et al.,

2008]. If the cost does not decrease since the latest remote control action, no new action is admitted until the terminal set has been reached. The effect is that the system exhibits and evaluates an unknown *dwell time*.

A consequence of this is that since there is always a valid return path to the terminal set, the terminal constraint can be removed from the optimization (12.4) when combined with (12.6), allowing a reduction of the control horizon N . Recurrent feasibility will not be ensured, but stability can be maintained through the recovery path. This could be important, since the addition of (12.6) can create a large optimization problem, increasing in size with N . On the negative side, the problem is at risk of often becoming infeasible, due to a short N , which will degrade performance.

Fewer recovery paths can also be considered. For instance, redefining Equation (12.7c) to

$$w_{j+1}^k = f_r(w_j^k, \kappa_2(w_j^k)), \quad k = 1 \quad (12.11)$$

modifies the MPC to evaluate a single recovery path. This is the path that will be used in the next time step, if the remote fails to update with new control input. The problem with this is that the controller is not recursively feasible. There is a good chance that the system ends up in a state that does not admit a new recovery path. When this happens, the client must start its graceful degradation which leads to reduced performance.

Implementation

The evaluation implements the scheme in an MPC with linear model, linear constraints and a quadratic cost function. The implementation, which is rather straight forward, is provided in this section. The controller was implemented using *quadprog* in Matlab [MATLAB, 2020] as *quadprog*($H, f, A, b, Aeq, beq, z_0$).

The matrix H and vector f set quadratic and linear costs. Only H is used, to form $J = x^T H x$, the linear term f is not used. x must contain the original states, recovery paths, and control inputs. It becomes

$$x = [z^T \quad (w^0)^T \quad \dots \quad (w^N)^T \quad u^T]^T \quad (12.12)$$

with $z = z_0, \dots, z_N$, $w^k = w_0^k, \dots, w_{N_r}^k$, $u = u_0, \dots, u_{N-1}$. The introduction of w^0, \dots, w^N into the optimization variables expand x to $x \in R^{N_x \times 1}$ and $H \in R^{N_x \times N_x}$, where $N_x = nN(2 + N_r) + mN$. Here, n is the number of plant states and m the number of inputs. All the new entries introduced by w^k in H are, however, zero, since the recovery paths do not introduce a cost penalty.

A and b are the inequality constraints, $Ax \leq b$. Every entry is of the form,

$$\begin{bmatrix} I_n & 0 \\ 0 & -I_n \end{bmatrix} \leq c_x(k) \quad \text{and} \quad \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix} \leq c_u(k) \quad (12.13)$$

where I_n is the $n \times n$ identity matrix and c_x and c_u are state and input constraints. Here, the constraints on z_k are replicated for $w_j^k, j = 0, \dots, N_r - 1$, and N terminal constraints, from \mathcal{T}_r , are entered for $w_{N_r}^k$.

In the equality constraints, $Aeq \cdot x \leq beq$, the matrix Aeq contains a mapping of z_k to w_0^k and a set of rows on the form

$$[\dots \quad -I_n \quad A - BK\Gamma \quad \dots], \quad (12.14)$$

expressing the action of the recovery mode in terms of the plant model A, B of the remote. On the r.h.s, beq depends on whether there is a defined recovery command. With no recovery command, all new entries in beq are zero. If there is a recovery command, it has to be considered that the initial state error, z_0 , is set in relation to command u_c as $z_0 = \hat{x} - \chi(u_c)$, while the error observed during recovery must be in relation to $\chi(\tilde{u}_c)$. To allow w_j^k to be set in relation to the origin, i.e., in relation to $\chi(u_c)$, the state transition during recovery is written as

$$w_{j+1}^k = (A - BK\Gamma)(w_j^k - x^*), \quad (12.15)$$

with the observed error, $x_e^* = w_j^k - x^*$, in relation to the recovery point x^* . The equation resolves to

$$-w_{j+1}^k + (A - BK\Gamma)w_j^k = (A - BK\Gamma)x^*, \quad (12.16)$$

where the l.h.s. is what was entered into Aeq above (Equation (12.14)) and the r.h.s. is constant. The recovery setpoint has to be translated in relation to the origin $\chi(u_c)$ to get

$$beq_i = (A - BK\Gamma)(\chi(\tilde{u}_c) - \chi(u_c)) \quad (12.17)$$

for all i relating to the recovery path. These are the necessary steps to implement the recovery control using *quadprog*.

In addition to *quadprog*, *lsim* and tools from *Jittertime* [Cervin, 2019] were also used in the implementation.










12.3 Evaluation

The following simulations show results from recovery enabled control implemented in the error signal generator framework. A brief reiteration of the setup follows.

Setup

The device controller is an ordinary LQR. The client implements an error generator function, (12.3), limiting the observed error in state x_1 . This keeps

Table 12.1 Configurations used in the evaluation. Results using the first six configurations are shown in Figure 12.3 and Figure 12.4. The last three are used for the example in Figure 12.6. #vars is the sum of control inputs u_k , predicted states z_k and recovery path states w_j^k , \tilde{u}_c is the recovery setpoint, \mathcal{T}_f is true if ordinary MPC terminal set is enforced. Q_Δ show the costs on x_1, x_2, x_3 in the recovery controller. Device costs are $Q_c = [800, 10, 1]$ and remote $Q_r = [1600, 10, 1]$

	N	N_r	#vars	\tilde{u}_c	\mathcal{T}_f	Q_Δ	Legend
C1	15	-	60	-	yes	-	
C2	-	-	-	-	-	-	
C3	15	5	285	-	yes	[1,100,10]	
C4	1	5	19	-	no	[1,100,10]	
C5	1	15	49	1.2	no	Q_c	
C6	1	15	49	-	no	Q_c	
C7	2	5	38	0.8	no	Q_c	
C8	5	5	95	0.8	no	Q_c	
C9	2	10	68	0.8	no	Q_c	

the plant and device controller within constraints. The remote controller is assumed to have a better model of the plant, other information, or more resources, allowing it to work with a different set of constraints. The remote also uses a different cost function, with more focus on reducing the error in state x_1 compared to the client. Using the remote (12.4) to control the plant, as opposed to the device controller, improves performance. Focus is on the feasibility and experienced performance loss when introducing the recovery. The terminal set \mathcal{T}_f and terminal cost of the remote MPC is derived by standard means from the device LQR, i.e., using the invariant set and asymptotic cost. The controllers execute at a rate of 10 Hz and the command signal u_c defines the plant setpoint $x_1 = 1.5$. The used configurations are listed in Table 12.1, with device and remote controller gains in the caption.

Performance without failures

Figure 12.3 show the effects on MPC performance when recovery paths are introduced. The plots show the three state variables. Colored areas show the constraints used in the remote (X_c , gray) device controller's extended state set (\mathcal{L}^+ , red) and remote terminal set (\mathcal{T}_f , yellow). The latter is an invariant set for (12.2) without (12.3). \mathcal{L}^+ is not shown for x_1 because it is defined to overlap with X_c . The terminal set \mathcal{T}_r is also excluded, and it equals \mathcal{L}^+ , thus requiring that the recovery ends within the client constraints.

In Figure 12.3, C1 shows the response of an ordinary MPC, which does not support failure recovery. The client response without remote control is shown

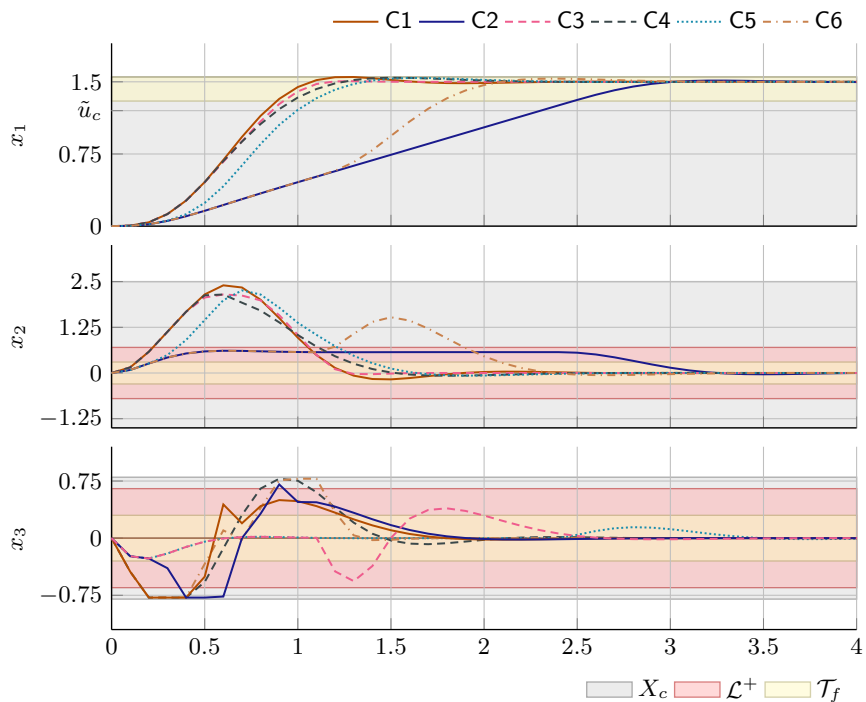


Figure 12.3 Examples of closed loop control with recovery guarantee. See configurations in Table 12.1.

in C2. This response is limited by the constraint on x_2 , implicitly ensured by the error signal generator. The performance of a recovery enabled MPC should fall between these two and preferably be close to the ideal response of C1.

Case C3 is close to achieving this ideal response. In this configuration, a recovery controller is defined by the cost Q_Δ , and is allowed five steps to enter the recovery terminal set. Q_Δ sets the objectives that are used to create K_r and generate Γ , as specified in R1 of Section 12.2. Under the conditions in Section 12.2 this controller is stable and recursively feasible, and therefore the nominal performance is guaranteed. The downside is the amount of necessary calculations. In order to implement the optimization, the problem must be extended with almost five times as many variables compared to the ordinary MPC, as seen in Table 12.1.

As was stated in Section 12.2, T_f can be excluded, allowing a feasible controller with a shorter N . In C4, T_f is removed and, to stress the point, N is reduced all the way down to one. This controller runs the risk of becoming

infeasible, in which case it must resort to a recovery path that will degrade performance. In the scenario, there is no noticeable failure and the controller achieves good performance, although not at the level of C3.

Reasonable performance can also be achieved using a recovery setpoint, here illustrated with $N = 1$ as C5. Notice from the configuration of C5 in Table 12.1 that the recovery horizon is extended to fifteen. This is necessary, to allow the device controller time to settle in the recovery terminal set \mathcal{T}_r . The recovery setpoint, \tilde{u}_c , is arbitrary, and the response of C5 could potentially be improved by moving \tilde{u}_c , possibly allowing it to vary over time.

The response in configuration C6 shows what happens if we go to the extreme, predicting only a single remote control action and not adding any recovery sequence. The performance is now limited by the device controller's response to the setpoint u_c , which must be ensured in case of failure. There is improvement, compared to the unassisted client, but the deviation from C2 happens late in the sequence. The remote controller evaluates the device controller's response using its own constraints and model. This allows it to override the error signal generator with an improved client response. The acceleration happens close enough to the setpoint, so that the client does not apply too much control action.

The take away from this is that, with the cost of a large optimization, it is possible to achieve a recovery controller with nearly ideal performance. Because the control response depends heavily on the recovery mechanism, it is also possible to retain much of this performance in other, less demanding, configurations. In the last two examples, increasing the horizon N will have little effect, because the response depends on the device controller's path to the recovery setpoint. Instead, without a specialized recovery controller, a large N_r is necessary, which also has the effect of a potentially long settling time for the recovery.

Performance with failures

Seeing in Section 12.3 that good performance can be achieved, we now look at the response in the event of remote failures. These responses are not compared to the ordinary MPC. Since the ordinary MPC does not provide any support for failure recovery, comparing to it could be misleading.

The examples in Figure 12.4 simulate external events causing failure. There are four event sequences, which are shown in Figure 12.5. From the bottom:

- 1) a single failure,
- 2) failure every other request,
- 3) an arbitrary random failure sequence, and
- 4) a failure over an extended period of time.

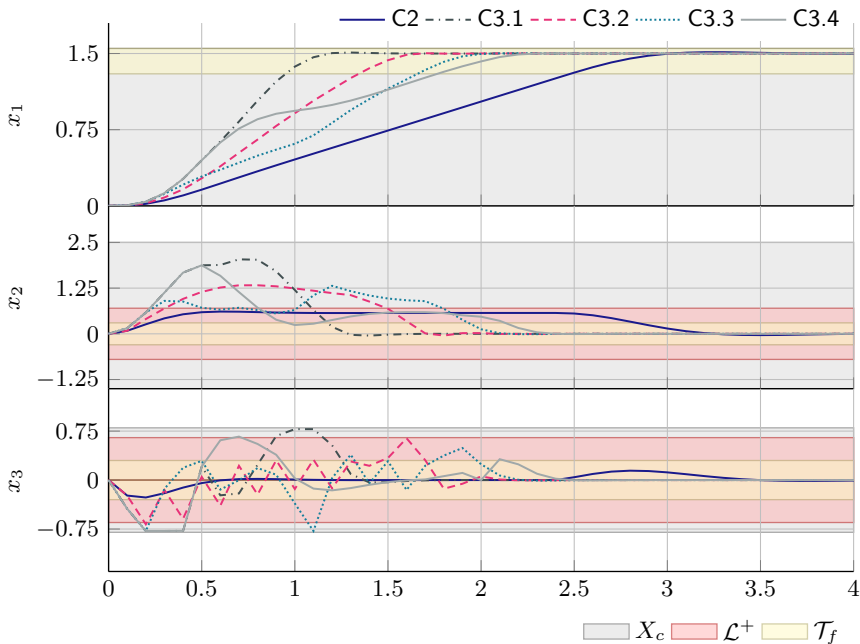


Figure 12.4 Examples of closed loop control with recovery guarantee when the system experiences failures due to the external events in Figure 12.5. See configurations in Table 12.1.

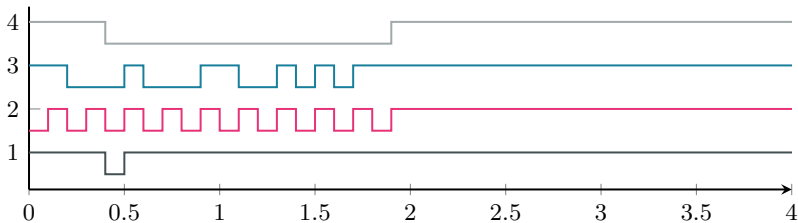


Figure 12.5 Fail sequences used for the simulation in Figure 12.4. When the signal is high there are responses from the MPC, when low there is a failure.

These sequences are applied to the high performing configuration C3. The second digit in the legend of Figure 12.4, i.e. d in C3. d , shows which sequence is active.

As seen from C3.1, the single failure has no visible affect on the response. When there is failure in every other request (C3.2) the performance degrades,

but it remains far better than the client. The recovery mode is not clearly distinguishable for states x_1 and x_2 , while there is a notable jagged behavior in x_3 . This is due to the recovery mode quickly trying to reduce x_2 and x_3 in every other sample. For C3.3 and C3.4 the mode changes are clearly visible in state x_1 . From C3.3 it is notable that performance recovers quickly when conditions improve, and from C3.4 that a short initial remote response is enough for notable performance improvement. In general, stability and efficiency are achieved also in the event of failures.

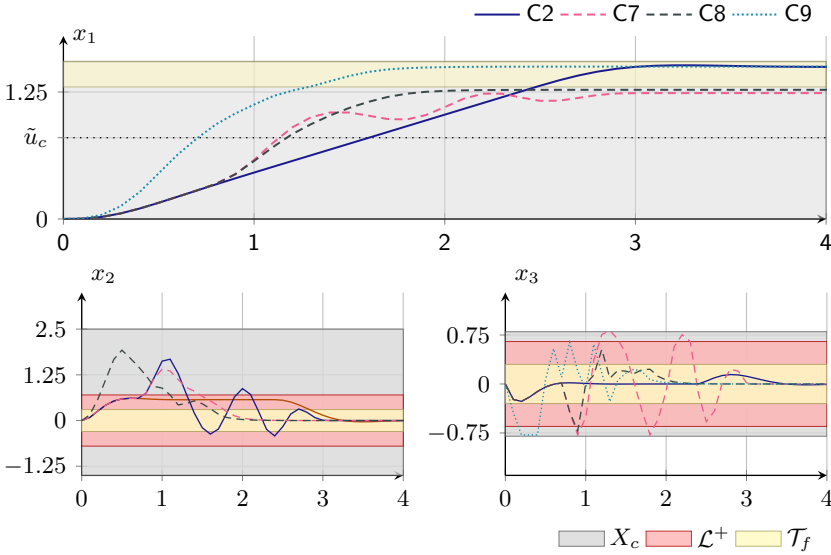


Figure 12.6 Examples of systems becoming infeasible and not reaching the setpoint.

Infeasibility and recovery setpoint limitations

The simulations are concluded with two cases where the controller provides an unwanted response, and an example remedy. This is shown in Figure 12.6. The client response, C2, is shown again for reference. The second configuration in the figure, C7, is a configuration that use short horizons, a recovery setpoint and the device controller for recovery. This configuration repeatedly applies too much control action and becomes infeasible. As a result, the performance is low and the setpoint is not reached with this configuration. With an increased N , the response of configuration C8 is better but the setpoint is still not reached. The combination of recovery controller (the unmodified device controller), recovery setpoint and short N_r reduces the reachable state

in x_1 . It would help to switch off the remote at some point, or move the recovery setpoint, allowing the state to proceed to a larger x_1 . Another remedy is to increase N_r , as in C9, providing an improved response.

It is worth noting that while C7 becomes infeasible and oscillates, it also stabilizes. With a less restrictive recovery setpoint it would have reached the intended state. If a recovery setpoint is used for the purpose of moving away from the constraint in case of failure, a large N_r may be necessary. Notice also the jagged line of C9 in the plot of x_3 . This implies that the controller remains affected by recurrent failure, which can also be discerned from the response in x_1 .

12.4 Conclusion

This chapter introduced an explicit recovery requirement into an MPC to handle uncertainty in the execution platform, and investigated its impact. An implementation using a feed-forward framework was shown. This provides a framework to extend a client controller and includes guaranteed recovery in case of remote failure, by manipulating input and output signals. It was shown that this can provide good performance and is reliable in the event of miss-configuration. In summary, a well defined recovery mode can provide a good controller response from a reliable client, and the explicit nature of the strategy allows for flexibility. The strategy is useful in best-effort control systems, where sub-optimal control is an acceptable trade-off, and where several options could be evaluated on-line. It also adds more calculations, and more variability among control clients. This will cause more load, and resource usage will be harder to predict. The final part of the thesis considers how to handle this load, without real-time requirements.

Part IV

Adaptation

13

Resilient Elastic Control

An approximate answer to the right problem is worth a good deal more than an exact answer to an approximate problem.

John Tukey

The following two chapters introduce a fundamental property of the cloud control system, in the interaction between the control application and the cloud. To appreciate how important this is, we first reiterate the aim of a single controller, in the context of the elastic cloud.

Figure 13.1 is an illustration of the predictive control problem and its various components, all of which have been presented earlier in the thesis. When this controller is implemented through the cloud, it opens up the po-

$$\phi(t) = \left\{ \begin{array}{l} \underset{\kappa}{\text{minimize}} V(x, \kappa(x, t)) = \sum^N J(x, \kappa(x, t), t) + \Omega(x_N) \\ \text{subject to } x(k + T_s) = f(x, \kappa(x, t), t) \\ x_k \in X(t), \kappa(x, t) \in U(t), x_N \in X_f(t) \end{array} \right\}, \dots$$

Figure 13.1 Variables in the definition of the MPC that may change at runtime, during development, and over the lifetime of the MPC controller. The number of requests symbolizes that several versions and instances may be evaluated in parallel.

This chapter is based on [Skarin et al., 2021]

tential to alter the various components on-line. Parameters or software may be changed by an operator, developer, or an automated process based on, e.g., machine learning or system health monitoring. The controller can be part of a plug-in scenario, in which components can be added to, or removed from the control problem. As components come and go, the control problem might be scaled and change parameters, requiring very different resources over time. Lastly, the control problem (or the client), can be implemented to adjust its requirements to meet current conditions. Chapter 10 provided an example of the latter, by implementing a variable horizon for the tracking problem. Chapter 11 provided another scenario, with controllers executing at different frequencies.

The following two chapters take a look at handling extensive processing times, network delay, and congestion, due to overload in the network or caused by extensive processing in a shared cluster. This provides further insight into the properties of the cloud control system. A generic method to handle congestion, both in terms of requests and in terms of individual computations, is to reduce the frequency of the controller. The goal is to reduce the load caused by the controller, while avoiding missed deadlines, and avoid entering a fallback mode. In this chapter, a single controller is considered and observations are made using simulated delays. In Chapter 14, the imposed delay is replaced by execution on real clouds, with real congestion, and controllers sharing a real cluster.

13.1 Background and related work

This section introduces two methods that enhance the performance of the cloud controller.

Execute to completion

As observed in various places and throughout this thesis, worst-case execution times of an optimizing controller, can be far from the median and average values. Since the worst-case execution times can occur at critical moments in the execution of the controller, these outliers cannot simply be discarded. Previously in the thesis, a single period delay has been used to manage the computational burden of the controller and delay from the network. In ordinary resource constrained scenarios, it is necessary to reduce the control frequency, allow sub-optimal control¹, implement a more efficient solver, or simplify the control problem, to ensure that the controller meets this execution time budget. In general, reliable worst case response times

¹Ending the optimization prematurely based on a deadline or maximum number of iterations.

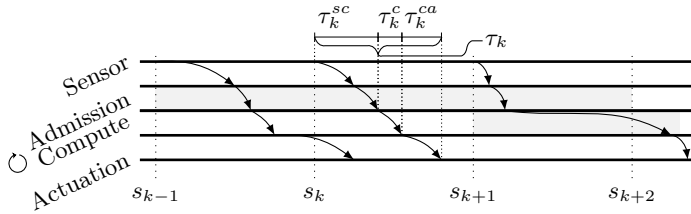


Figure 13.2 Delay constituents of the serviced CCS. Our primary concern in this chapter is with the admission and processing times.

pose a problem in control design [Findeisen and Allgöwer, 2004; Tobuschat et al., 2016] and lead to conservative choices. Soft real-time is one approach to this problem, but it introduces its own issues with lateness [Fontanelli et al., 2013]. Others have suggested improving overall performance through probabilistic approaches to worst-case execution time [Bernat et al., 2002], and the continuous stream model [Fontanelli et al., 2013] to remove dependence on previous jobs. In an embedded system, it remains important that the system does not become overloaded, but in the cloud service model, requests can execute in parallel. Sporadic overruns of the execution time budget does not overload the cloud system. If the average or peak load is too high, it can be assumed that the cloud will be reconfigured accordingly, through vertical or horizontal scaling. Therefore, the elastic controller can choose to execute all requests to completion and make an informed decision when results arrive.

Resource constraints

In practice, the remotely supported cloud controller can, and will, run into resource constraints, for several reasons. In a deploy-anywhere scenario, the controller may be deployed to a limited edge device. In the cloud, the controller may execute in an overloaded cluster. Communication to the cloud and performance of the client (in managing all request and responses), can also be a bottleneck. Constraints may arise in the network delay and bandwidth, CPU and memory, number of worker nodes, the performance of load balancing strategies etc, and the resilient controller must handle them all.

Figure 13.2 provides an updated view on the delay constituents from Section 3.1. This forms a view of the delay when a control decision is requested from a cloud service. The figure adds admission time and replaces the control decision by a computation. They are further accentuated by the circled arrow to the left, illustrating that these components may arise from a chain of events. Admission represents routing and forwarding to a worker node in the cloud, while the computation is the actual work performed for the controller. This forms the cloud portion of the experienced delay. The sensor to

admission delay and the compute to actuation delay represent the network that separates the client from the cloud service. The important take away from this figure is that each of these components can be affected by external factors, but also by the frequency of requests, and by the configuration of the individual control problem. A further complication is that many components can affect each other in a cluster, where even the admission may, directly or indirectly, be affected by the controller function and vice versa².

There are works that propose and study real-time closed loop control over the cloud [Lyu et al., 2019; Pelle et al., 2019; Heilig et al., 2015], and suggestions to use advanced methods, such as stochastic predictive control [Esen et al., 2015], to handle latency. While researchers have not been oblivious to the elastic nature of clouds, the view of a traditional NCS remains. This leads to no consideration of, for instance, transitional periods of scaling. An adaptive, *best-effort* system can arguably be made very efficient working with the cloud, and also supports the deploy-anywhere scenario. A generic way to gracefully degrade the control system is to change control frequency, as has previously been used for handling network quality of service [Björkbom et al., 2010], bandwidth sharing in sensory networks [Xia and Zhao, 2007], and overload scenarios in real-time systems [Buttazzo et al., 2007]. This is often also referred to as sample rate adaptation, but since the sampling rate on the client might not change, it is referred to as frequency control here.

Because of significant uncertainty and variability, delay mitigation and adaptation are of particular interest to control over public and wireless networks [Kim et al., 2006; Yang et al., 2005; Ploplys et al., 2004]. Recent works target modern concepts such as industrial Internet-of-things (IIoT) devices and Fog computing [Inaltekin et al., 2018], and adapting to sporadic overruns in off-the-shelf embedded control systems [Mubeen et al., 2017]. When adding the elasticity aspect, a frequency modification can improve the controller response, but it also relieves the cloud from some of the load, giving it time to scale. Three useful concepts are therefore combined:

- 1) executing to completion, removing execution time deadlines for control calculations,
- 2) client performance scaling, modifying the control performance of the client to support transitional periods and deploy-anywhere, and
- 3) cloud resource scaling, providing the benefits of elastic computing.

²As seen in Figure 2.3 many cluster functions can execute in the shared environment. Compare also the HAProxy and Kubeless deployments in Figure 8.7

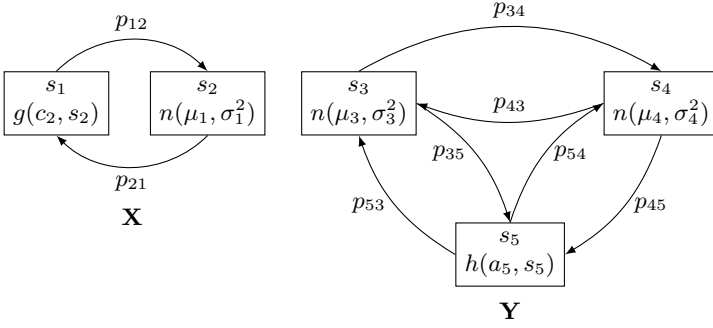


Figure 13.3 Markov processes for processing time (\mathbf{X}) and flight time (\mathbf{Y}). Transitions back to the same state have been excluded for brevity.

13.2 Simulations

This section describes how delays are simulated using Markov processes and theoretical distributions, fitted from observations. Further, it presents how the control loop in this chapter is subject to setpoint changes and random disturbances.

Markov processes

Delays are obtained from the two Markov processes in Figure 13.3. The figure shows states as rectangles. Inside a rectangle there is a state name, s_i , and a distribution function, g, n, h . The edges, p_{ij} , shows transitions from state i to j , which occurs with a certain probability. The transitions back to the same state, i.e., p_{ii} have been excluded to simplify the figure. For each step in the simulation, transitions are evaluated and delays are then drawn using the distributions specified by the new states. \mathbf{X} and \mathbf{Y} represent the random variables that draw from these two processes. The former, \mathbf{X} , is used to calculate the processing time of a request and the latter, \mathbf{Y} , provides the flight time. The processing time of request ϕ_k is obtained as

$$\tau^c(\phi_k) = i(\phi_k)\mathbf{X}(k)\frac{N(\phi_k)}{N_{ref}}, \quad (13.1)$$

where $N(\phi_k)$ provides the prediction horizon in the request, and $i(\phi_k)$ provides the number of iterations required in the optimization. N_{ref} is the reference horizon used to generate the data sets³. Thus, the random variable \mathbf{X} provides a measure of the processing time per iteration, scaled by the horizon. \mathbf{Y} provides the flight time without modification.

³The used data was produced with $N_{ref} = 10$.

Table 13.1 Transition probabilities in two different scenarios. Values are given in a condense scientific notation, where $x^{-y} = x \cdot 10^{-y}$. Transitions are evaluated in every step of the simulation, resulting in 200 Hz times every simulated second.

Scenario 1		Scenario 2		Scenario 1 and 2					
p_{12}	p_{21}	p_{12}	p_{21}	p_{34}	p_{35}	p_{43}	p_{45}	p_{53}	p_{54}
1^{-3}	1^{-3}	.75	.25	9^{-5}	1^{-5}	2.5^{-4}	2.5^{-4}	3.5^{-4}	1.5^{-4}

Two scenarios are defined. The flight time process remains the same in both scenarios, but the processing time probabilities change. The transition probabilities are shown in Table 13.1. In Scenario 1, the states s_1 and s_2 represent an under- and an over-utilized state. To create a reasonable load, the processing times are scaled so that the under utilized mode, s_1 , execute with an average load of 60%, assuming a single machine, over a period of ten seconds. This must be considered low, since one aim of the shared cloud is to achieve high utilization. In effect, from the distributions in the next section, s_2 will execute well above full utilization, thus requiring several workers to handle the optimizations to avoid queuing requests. On average, it takes several seconds before switching to the other state, but over time they are equally used.

In Scenario 2, the states switch frequently with a higher probability of being in state s_2 . In this mode, there is a high probability of transition every time X is drawn. The transition probabilities were calculated from measured data. In addition, when executing this scenario, it is assumed that there is one request processing queue for each of the two modes. In contrast, Scenario 1 simulates either a single processing queue or no processing queue at all (i.e. allowing any number of parallel requests).

The flight time states s_3 and s_4 resemble a nearby data center with ordinary, fast response (s_3) and a degraded response (s_4). State s_5 represents another, more distant data center. The transition probabilities provide recurrent entry to the degraded state, and less frequently switching to the remote data center. The probability of switching to s_5 goes up in the degraded state, and when returning, it is more likely to enter s_3 than s_4 . This is an ad-hoc scenario, where s_5 is used to offload the primary data center. The primary purpose of the flight time modes is to observe if a change in network delay, such as it has been registered towards the data centers, imposes a large detrimental effect on the frequency control. The request rate of the controller does not affect the flight time distribution, while in practice, congestion in the network and admission will also be mitigated by the adaptive controller.

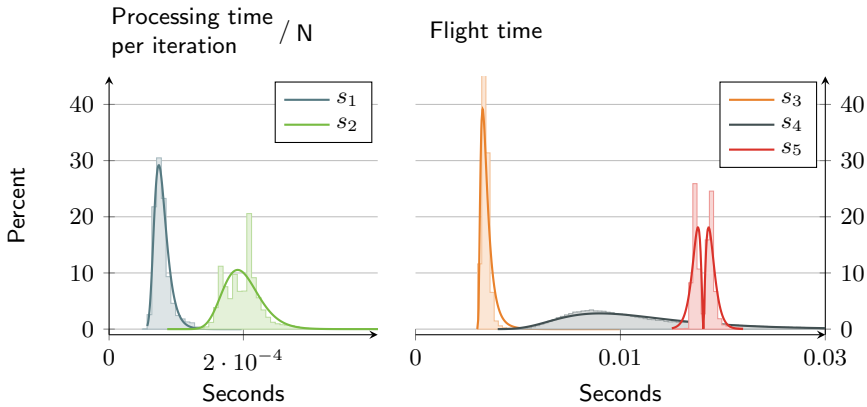


Figure 13.4 Maximum likelihood fitted distributions for the states in Figure 13.3

Distributions

Each state in Figure 13.3 provides a different random distribution for either the processing time or the flight time. These distributions, and the parameters in Table 13.2, were obtained by fitting data through maximum likelihood estimation (MLE), using the SciPy library [Virtanen et al., 2020]. The distributions have been chosen because they fit the data well, not because of any sought properties. Histograms of the input data, overlaid with the used distribution, is shown in Figure 13.4.

The left, processing time, graph in Figure 13.4 shows two distinct modes. These were registered towards a single cluster in ERDC. Exactly what causes these modes is not known but it was concluded from examining the data that 1) the modes do not appear distinct in time, and 2) they do not repeat with any obvious pattern. Rather, requests seem to fall randomly into one of the modes, with s_1 three times as frequent as s_2 . This observation is represented by Scenario 2.

Table 13.2 Parameters for the distributions, and the offset of the random variables (i.e. b in $f(x - b)$), for states of Figure 13.3

State and Distribution			Parameters		Offset
s_1	gen. logistic	(13.2)	$c = 946$	$s = 8.91 \cdot 10^{-6}$	$1.3 \cdot 10^{-5}$
s_2	lognormal	(10.13)	$\sigma = 0.193$	$\mu = -8.86$	$5.53 \cdot 10^{-5}$
s_3	lognormal	(10.13)	$\sigma = 0.632$	$\mu = -7.14$	$6.02 \cdot 10^{-3}$
s_4	lognormal	(10.13)	$\sigma = 0.431$	$\mu = -4.2$	$5.56 \cdot 10^{-3}$
s_5	double gamma	(13.3)	$a = 2.52$	$s = 3.4 \cdot 10^{-4}$	0.0281

The flight time data for s_3 and s_5 , on the right side of the figure, is from the Kubeless deployment in Chapter 8. Flight times for s_3 were registered towards the smaller, municipal data center, while s_5 is from the larger, distant data center. There are two clear modes in the data for s_5 . A single distribution is preferred, and the double gamma distribution fits well with the observations. Finally, s_4 is a refit of the distribution for case β_2 in Section 10.3, in a different time scale. We know from Chapter 10 that this represents a case when the flight time experiences large variations due to load. These distributions are entered into the states of Figure 13.3, as suitable random number generators.

There are three distributions, $n(\mu, \sigma^2)$, $g(c, s)$, and $h(a, s)$. The function $n(\mu, \sigma^2)$ represents the log-normal distribution, which was also used in Chapter 10, with probability density function specified in Equation (10.13). The function $g(c, s)$ represents the generalized logistic distribution function, with the probability density

$$g(c, s) : f(x) = \frac{c \cdot e^{-x/s}}{s(1 + e^{-x/s})^{c+1}}. \quad (13.2)$$

and $h(a, s)$ the double gamma distribution,

$$h(a, s) : f(x) = \frac{1}{2\Gamma(a)} |x/s|^{a-1} e^{-|x/s|}, \quad (13.3)$$

where

$$\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt.$$

Tracking and disturbance

The simulations execute periodic setpoint changes and introduces significant pulse disturbances into the plant state. Setpoints are as usual close to the constraints. Disturbances, and errors, can force the system to break constraints, but soft constraints (see next section) allow the optimization to remain feasible. Disturbances enter the system as

$$x(k+1) = Ax(k) + Bu(k) + [0 \quad w(k) \quad 0]^T, \quad (13.4)$$

i.e., adjusting the speed of the ball. Here, A and B are the state space matrices for the base frequency, discretized using $h_q = 0.005$. The sequence of disturbances is,

$$\mathbf{w} = [0 \quad \dots \quad w_0 \quad 0 \quad \dots \quad w_1 \quad 0 \quad \dots], \quad (13.5)$$

where the values of w_i are drawn from a normal distribution

$$\mathcal{N}(\mu, \sigma^2) : f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad (13.6)$$

with $w_i = \mathcal{N}(\mu_w, \sigma_w^2)$, $\mu_w = 0$, $\sigma_w^2 = 0.09$. The position in \mathbf{w} , i.e., the time step k , is obtained as $k(w_i) = \lceil t(w_i)/h_q \rceil$, where $t(w_i)$, is drawn using another normal distribution

$$t(w_i) = \mathcal{N}(\mu_d, \sigma_d^2) + t(w_{i-1}). \quad (13.7)$$

with $\mu_d = 2$, $\sigma_d^2 = 0.25$. The time between two events varies but on average, disturbances occur with a distance of two second.

13.3 Offloaded controller

The client in this chapter and the next, solves the control problem

$$\underset{\mathbf{z}}{\text{minimize}} \quad V(x_0, T_s(t)) = \sum_{j=0}^{N-1} l(z_j, T_s(t)) + V_f(z_N, T_s(t)), \quad (13.8a)$$

$$\text{subject to} \quad x_{j+1} = f(x_j, u_j, T_s(t)), \quad (13.8b)$$

$$h(u_j) \leq 0, \quad (13.8c)$$

$$g(x_j, \phi_j) \leq 0, \quad (13.8d)$$

$$z_j = [x_j^T \quad u_j^T \quad \phi_j^T]^T, \quad (13.8e)$$

$$x_0 = \hat{x}(x(t), \mathbf{u}(t), x_s, \mu(t)), \quad (13.8f)$$

$$N = \lceil N_s/T_s(t) \rceil. \quad (13.8g)$$

Equation (13.8) does not include a terminal set but instead implements soft constraints, through the slack variables, ϕ_j . Note that this is different from the examples in the previous chapters. The problem is parameterized using the time varying sampling period $T_s(t)$, and when implemented, Equation (13.8) is discretized as necessary using the procedures in Chapter 11.

The controller horizon is defined as a time period N_s , in seconds, and converted to discrete time steps in (13.8g). The value is rounded up to ensure that the prediction time of the controller is always at least N_s . The initial state x_0 is defined as a function that depend on the current state, the selection of control signals $\mathbf{u}(t)$, the setpoint x_s , and prediction steps μ . \mathbf{u} and μ are expressed in a base time step, which is implemented as a factor of $T_s(t)$. Note that t is constant in Equation (13.8), and only used to initialize the values in the optimization problem. The generalization of the initial value as a function allows the client to implement the predictor in Equation (3.3). The value of $x(t)$ is obtained through state feedback or an observer (Equation (3.2)).

The client structure is shown in Figure 13.5. The MPC block represents Equation (13.8), on which the client depends for tracking the setpoint, u_c . The predictive part is included in this block. To the right of the MPC, there is a frequency controller block. A *loss measure*, ρ , is input to this block. It

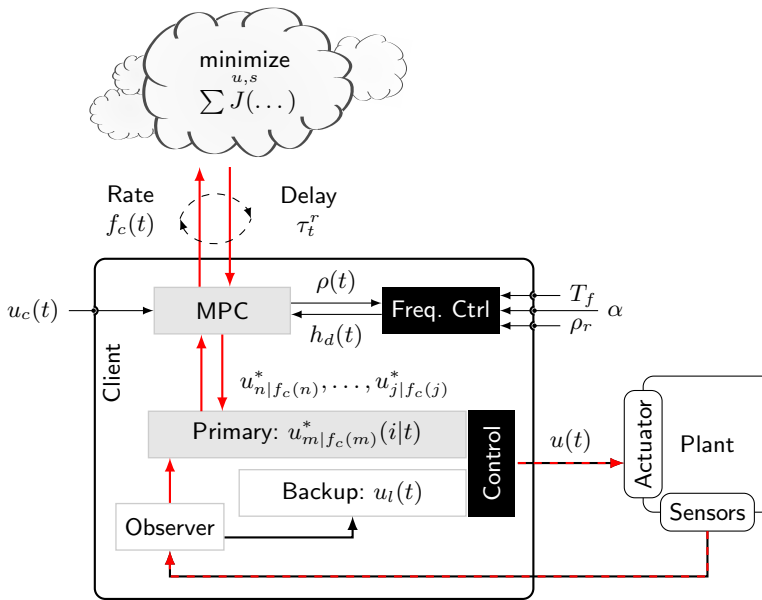


Figure 13.5 Client structure, with a frequency controller as input to an optimizing controller in the cloud, and a backup controller.

outputs h_d , which translates to $T_s(t)$ in Equation (13.8). The output from the MPC block is input to the primary control block that selects from the results. If the primary control block fails to provide a control action, the client falls back on a backup controller. Before falling back on this controller, the client will use data from Equation (13.8) in open loop. For the reference plant, the client implements a fallback controller that regulates it back to the origin. This can be alternatively replaced by, for instance, the explicit recovery that was presented in Chapter 12, but the use of open loop and implementation of frequency control must be handled with care. A conservative backup is used here, implemented as an LQR, to clearly observe the effects of delay on (13.8) and simplify implementation.

Dead time, control selection and open loop

Reusing the nomenclature from Chapter 10, the request set, ϕ_k , is a single request sent every sampling period, T_s . The response set, ψ_k , is collected continuously and can contain responses in any order. Events are synchronized by the time slotted system illustrated in Figure 13.6. The rows in this figure show MPC results. Horizontally, the indexed slots represent predicted control actions. Filled slots are used, with the applied control actions shown below

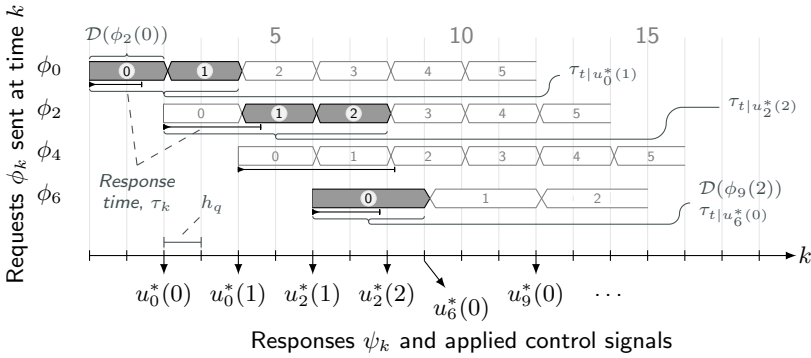


Figure 13.6 Time slotted selection of controller result. Dark slots have arrived within their deadline and are used. The frequency is changed at $k = 6$. $\mathcal{D}(\phi_k(i))$ is the imposed dead time of a request and $\tau_{t|u_k^*(i)}$ the effective control latency when using control action i of response ψ_k .

the x-axis. The first three requests are executed with a control period of $T_s(t) = 2h_q$. The fourth, modifies this period to $T_s(t) = 3h_q$.

As previously, the controller uses a fixed control delay of one sampling period. $\mathcal{D}(\phi)$ is introduced to represent the dead time of a request. The time index k represents steps in the base time period h_q , and μ is an integer number of steps in this discrete time. Because no more than one request is sent at the same time instant, responses can be referred to using their request time, i.e., ψ_k is the single response to the request ϕ_k . Equally, the resulting control actions are given as u_k^* , the response time as τ_k and so on.

The controller achieves closed loop mode when $\mathcal{D}(\phi_k) < \tau_k$, in which case $u_k^*(0)$ can be applied. When the closed loop response fails to arrive, the controller applies open loop actions from the latest selected response. In general, with j the time index of the latest selected response, the controller applies

$$u(t) = u_j^*([\!(t - t')/T_s(\phi_j)\!]), \quad (13.9a)$$

$$t' \leq t < t_j + \sigma, \quad (13.9b)$$

$$t' = t_j + \mathcal{D}(\phi_j), \quad (13.9c)$$

where t' is the predicted time for the first control, $u_j^*(0)$. $T_s(\phi_j)$ provides the sampling rate of request ϕ_j , and σ is the *maximum control latency*. σ is given in seconds and is independent of the frequency. The backup controller is used when $t < t'$ and $t - t_j > \sigma$, independent of whether there are more control actions in u_j^* (the number of which is determined by the independent controller horizon N). Thus, there is a fixed maximum time from sampling

to actuation. The number of available open loop control actions is thereby reduced with the frequency.

Returning to Figure 13.6. On the first row, the response to request ϕ_0 arrives in time to apply closed loop control. On row two, ψ_2 arrives late, as seen from the response time and that $u_2(0)$ is not used. However, control actions from this response are applied, starting with $u_2^*(1)$. The third request, ϕ_4 , is further delayed. It is registered by the client, but not used. Instead, a frequency switch happens at time $k = 6$, on the fourth row, and $u_6^*(0)$ is applied at $k = 9$. The controller always selects the latest available request and the open loop procedure is used to fill in the gaps. Note that both $u_2^*(1)$ and $u_2^*(2)$ were included in the prediction as

$$x_0(k+3|k=6) = A(A\hat{x}(k) + Bu_2^*(1)) + Bu_2^*(2), \quad (13.10)$$

to form the initial state of ϕ_6 . Here, A and B represent the discrete model based on h_q , and \hat{x} is the observed state.

Notes on open loop and predicted delay

If the controller is stable, we should not have to be concerned with an open loop path in the nominal sense. However, this requires that the control sequence starts with the first control action, $u^*(0)$. In Figure 13.6, the change from ψ_0 to ψ_2 discards this. A difference between the control actions $u_0^*(1)$ and $u_2^*(0)$ will cause $u_2^*(1)$ to not implement the predicted optimal control. From the basic stability argument, this is a problem, but in practice it is useful to apply ψ_2 . If a disturbance or setpoint change happens, it will be acted upon quicker. It also avoids going into the backup mode. Improvements can be made by compensating for the missed control action, but that is not considered here.

Deadline

Previous chapters have used a one sampling period deadline mode. In that mode, the response to ϕ_2 in Figure 13.6 would have been discarded. If the response ψ_0 includes the open loop prediction, the client would continue to apply results from ψ_0 for as long as σ allows. The benefit of this mode, is that it behaves like a hard real-time mode, and that it can relieve the remote from extensive calculations. A further limitation can be to not transfer the open loop sequence, i.e., replacing Equation (13.9b) with

$$t' \leq t < t' + T_s(\phi_j) = t_j + \mathcal{D}(\phi_j) + T_s(\phi_j) \quad (13.11)$$

to obtain a forced closed loop mode. In this case, the remote only has to transfer a single control action over the network, with the downside that steps $k = 2, \dots, 9$ in Figure 13.6 would have to execute the backup controller.

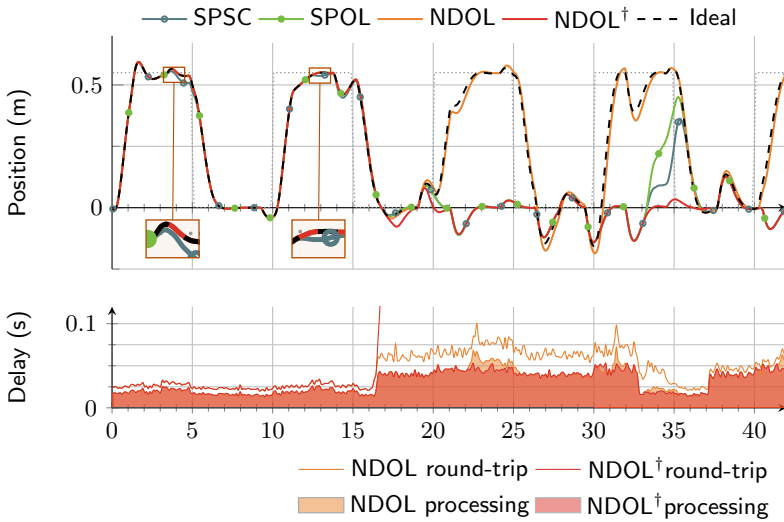


Figure 13.7 Trajectory and experienced delays of non-adaptive configurations in Scenario 1, described in Section 13.2. The dotted line is the setpoint. All controllers run at 33 Hz. See Table 13.3 for definitions of the modes.

Five modes are defined in Table 13.3. In addition, as mentioned in Section 13.2, simulations can either assume parallel execution with independent delays, or that requests must wait for previous processing to complete. The † is appended to results when the latter applies. In these modes, Scenario 1 executes with a single queue, while Scenario 2 has one queue for each state s_1 and s_2 .

Figure 13.7 shows the first four modes in Scenario 1, illustrating the combination of execute to completion, and open loop delay mitigation. The upper graph shows the response in the tracked position state. The graph below shows experienced delays for the NDOL mode with and without independent delays. The plot contains round-trip times as solid lines and processing times as filled areas. Relating this graphs to the modes presented in Section 13.2, the first part of this sequence executes in mode s_1 and s_3 , representing the ordinary, good conditions. At sixteen seconds, the flight mode changes almost simultaneously with the processing time mode, to s_2 and s_4 . At this point, the flight time of NDOL† disappears of the graph, due to the build up of processing queues. Later in the sequence, at 33 seconds, processing times improve and the flight time soon also reduces. A few seconds later, the processing times increase again. The NDOL† will eventually recover, but it takes a long time, since no requests are discarded.

Table 13.3 Client modes

Ideal	Result of the MPC assuming that all requests respond within one sampling period, i.e., $\tau_k < T_s \forall k$.
SPSC	Single Period Single Control. Transfers only the first control action, $u_k^*(0)$, to the client, and has a request deadline of one sampling period, i.e., implementing Equation (13.11).
SPOL	Single Period Open Loop. Transfers the complete result of the optimization and has a request deadline of one sampling period. Open loop is used to mitigate delay (until the control latency reaches σ). Implements Equation (13.9) with the requirement $\tau_k < T_s$.
NDOL	No Deadline Open Loop. Transfers the complete result of the optimization and has no request deadline. Applies open loop until the control latency reaches σ .
Adapt	The controller in Figure 13.5, with no request deadline. Applies open loop until the control latency reaches σ .
†	When † is appended to a mode, the remote system can become overloaded and requests queued, i.e., the simulation does not assume independent delays. This applies to NDOL and Adapt but makes no difference to SPSC and SPOL, since requests in these modes have a single period deadline.

In the initial, favorable conditions, all modes are able to complete the objective fairly well. However, the SPSC mode must sometimes resort to the backup controller, as seen in the marked locations around four and thirteen seconds. This shows that there are delays causing deadline misses. These delays are handled by the second mode, SPOL, which enforces a deadline, but uses open loop to mitigate the loss. The mode without deadline, NDOL, also executes perfectly, following along the dashed path of the ideal response. When the conditions drastically deteriorate, at sixteen seconds, the modes SPSC and SPOL fail, and resort to execute only the backup controller. A delayed response is now also observed for the NDOL mode, showing a sluggish behavior as a consequence of the control latency. Nonetheless, by executing to completion and applying the resulting output, this mode continues to meet the objective. This is good, but part of a simulation that is assuming independent delays. Because NDOL does not have a deadline, requests can pile up, as for NDOL[†], and this will cause the client to enter the backup mode.

Scenario 2 is shown in Figure 13.8. A third plot is added to this figure, which will be returned to shortly. The dominating source of delay in these simulations is processing, and because Scenario 2 is alternating between s_1

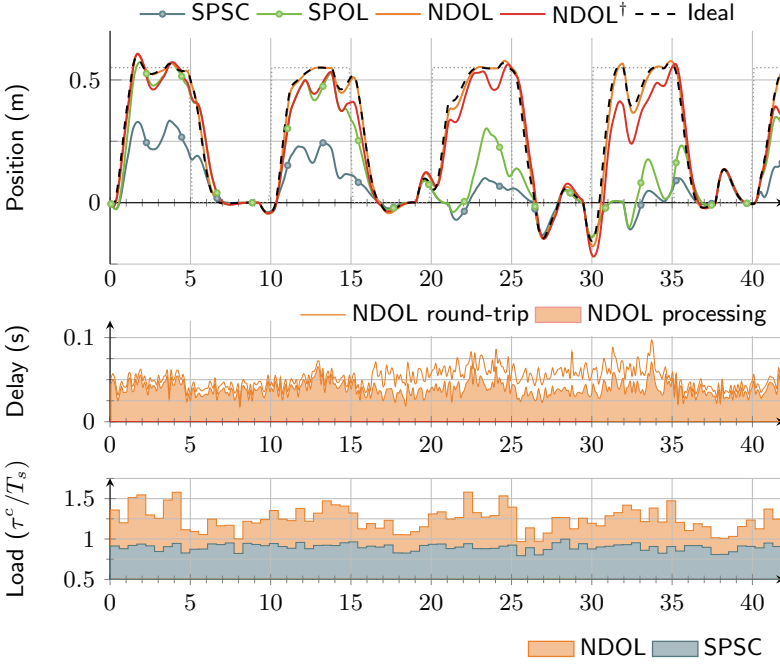


Figure 13.8 Trajectory and experienced delays of non-adaptive configurations in Scenario 2, described in Section 13.2. The dotted line is the setpoint. All controllers run at 33 Hz. See Table 13.3 for definitions of the modes.

and s_2 , the delay is now more even. Compared to Scenario 1, this has a clearly negative effect on all cases except NDOL in the first part, up until sixteen seconds. After sixteen seconds, there is instead a positive effect. This does not make SPSC or SPOL useful, while NDOL[†] now manages to reach the setpoint. What might not be directly obvious, is that the result for NDOL[†] in effect comes from inefficient use of mode s_1 . This is the reason for adding the third plot, showing the average processing load (as a mean over 0.5 s). The NDOL data shown here, is really just a scaled version of the processing data in the delay plot.

First, we look at SPSC. Since SPSC has a single period deadline, it will never execute with a load above one. Requests that experience delay drawn from s_2 will fail, unless they finish in very few iterations. A request will for sure not be returned unless

$$\lceil N_s/T_s \rceil \cdot \mathbf{X} \cdot i < T_s, \quad (13.12)$$

where i is the number of iterations required in the optimization. With $N_s =$

1, $T_s = 0.03$, and $\min(\mathbf{X}|s_1) \approx 1.5 \cdot 10^{-4}$ (Figure 13.4), i can never be larger than five. SPSC is therefore roughly equivalent to a 25% chance of getting a response, as successful results mostly come from s_1 . When the network delay is increased, this is reduced further as the fast processing mode also often fails to provide responses. The execute to completion mode, NDOL, gets many delayed responses, as seen from the load, but with no major consequence. This is seen from its trajectory closely resembling the ideal dashed line. However, the system is overloaded, and in NDOL[†], mode s_2 will build up a processing queue. Again, the system will be driven by responses from s_1 . Because NDOL[†] accepts late responses, it manages this better than SPSC.

The situation of NDOL[†] in Scenario 2 is due to the low performance of state s_2 . A potential remedy, is to send more requests to mode s_1 , although it is not known in practice, if this will cause it to behave more like s_2 . Another solution is that the cloud can identify the load problem and scale, turning the situation from NDOL[†] into NDOL. A third mitigating strategy is to use frequency adaptation in the client. This modifies T_s in Equation (13.12), having an effect on both sides of the equation. Therefore, doubling the sampling time to 0.06 will go from five, up to 23 iterations in the optimizations. The next section shows results from simulations that use frequency adaptation in combination with execute to completion. Details of the implementation are presented in Chapter 14.

Frequency control

The frequency control adjusts the sampling rate of Equation (13.8), based on a measure of loss. In the following examples, this loss is related to the experienced delay, as detailed in Chapter 14. The focus is on executing requests to completion, but it is easy to apply the frequency control also to a system that enforces the single period deadline. The derived controller, and details on how the loss factor is calculated, are presented in Chapter 14. Here, we first look at some results from the adaptive controller in the simulations.

A Proportional and Integral (PI) controller is used for frequency control. The setpoint of the frequency controller is at 5%, and it works at a sampling rate of 100 ms. Figure 13.9, is illustrating the results from this frequency control. The figure combines the two scenarios from Table 13.1 (the same scenarios that were used in the previous section). Figure 13.9 again shows the balls position at the top and the computational load at the bottom. The experienced delay has been replaced by a plot of the selected frequency. In the upper graph, it is immediately noticeable that the adaptive controller works well. Drops in disturbance rejection performance can be observed in some places (for instance the marked occasions at 14 s and 30 s), but the overall response is good in all cases. The controller adjusts frequency, translating

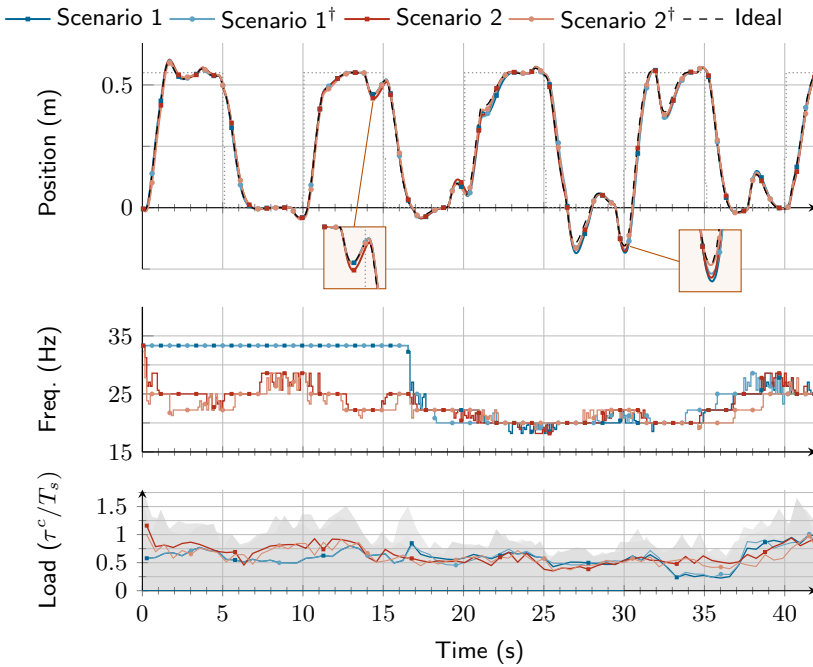


Figure 13.9 Trajectory and experienced delays of the adaptive client for Scenario 1 and 2, described in Section 13.2. The dotted line is the setpoint.

into reduced processing times and improved results. The frequency adjustment is shown in the middle graph, and the processing load at the bottom. There is a gray back-drop in the load graph. This shows the maximum load, and the solid lines show the average (again as the mean over 0.5 s). While the average load is below one, throughout the graph, it is clear from the background that there are occasional processing time overruns. This is adjusted and compensated for by the frequency controller. When the system registers loss, it will quickly compensate by adjusting its frequency. This is seen from the frequency variations of the two configurations in Scenario 2. These variations are larger at the higher frequencies, from 0 s to 16 s and after 37 s. It can also be observed that in Scenario 2, the † mode often works at a somewhat lower frequency, due to that the service can be overloaded.

These changes in frequency keeps the average load fairly stable. There is a distinct drop in load for Scenario 1 from 32 s to 33 s. At this point, there is a mode change, which is clearly visible in the delay graph of Figure 13.7. Between 34 s to 35 s, the frequency controller starts to increase the frequency, and an increase in the average load follows. In Scenario 1, the processing

delays increase again at 37s, otherwise the frequency would soon return to 33 Hz. Also in Scenario 1, a change in processing times is causing a large drop in frequency at 16.5s. This change is visible in the load graph, as a small peak in the average load. If the frequency response was slower, the average load would surpass one, causing queued requests in Scenario 1[†], and likely notable latency issues.

A final thing to notice in this figure is that the frequency and load goes down, and frequency changes become smaller, for Scenario 2 at 16s. Conditions return towards the end of the timeline. This is attributed to the additional transmission delay that happens between 16s to 35s (Figure 13.8). This delay is compensated for, but is not affected by the load from the client. Nonetheless, the controller will better predict the outcome of requests and reduce the overall response time. The primary takeaways from this figure is that the used frequency control handles load changes well, and that changes in the frequency do not display any clearly negative effects.

For a more detailed analysis of the performance, the simulation is executed over an extended period. In this longer sequence, all combinations of modes (Figure 13.4 and Table 13.2) are represented. A performance measure must also be defined, and the control cost is not suitable. The reason is that the simulations combine step changes with disturbances, and when the backup controller is used, it can score a good control cost. This is not the intended result, since use of the backup controller should be avoided. To be clear, consider a client that must consistently use the backup controller. This client will keep the state $x_1(k)$ close to zero. Let the setpoint be distinctly non-zero, $|x_{sp}(k)| \gg 0$. A functional remote controller would keep its plant state, $\bar{x}_1(k)$, close to the setpoint, $\bar{x}_1(k) \approx x_{sp}(k)$. If now the setpoint is changed to zero, the client using the backup controller obtains a good score due to that $x_1(k+1) \approx x_{sp}(k+1)$, but the correct state of the system is close to the previous setpoint, represented by $\bar{x}_1(k+1) \approx x_{sp}(k)$.

A comparison is therefore made towards the ideal controller response, introduced as the closed loop response error (CLRE), and defined as

$$\text{CLRE}(t) = \int^t (x_1(s) - \bar{x}_1(s))^2 ds, \quad (13.13)$$

where x_1 is the value of the measured response, and \bar{x}_1 is the value of the ideal response (represented by the dashed line in the previous figures). Further, the control responses in Figure 13.7 and Figure 13.8 execute at 33 Hz and it is already known that they cannot handle the case, [†], of dependent processing delays. The result from Figure 13.9 indicate that a frequency around 20 Hz might work well, and 22.2 Hz is chosen in an attempt to beat the adaptive controller. To also see if it is possible to obtain a better result by simply using an even lower frequency, a 16 Hz controller is also compared. Note that the adaptive controller can go lower (down to 10 Hz). The goal is not to motivate

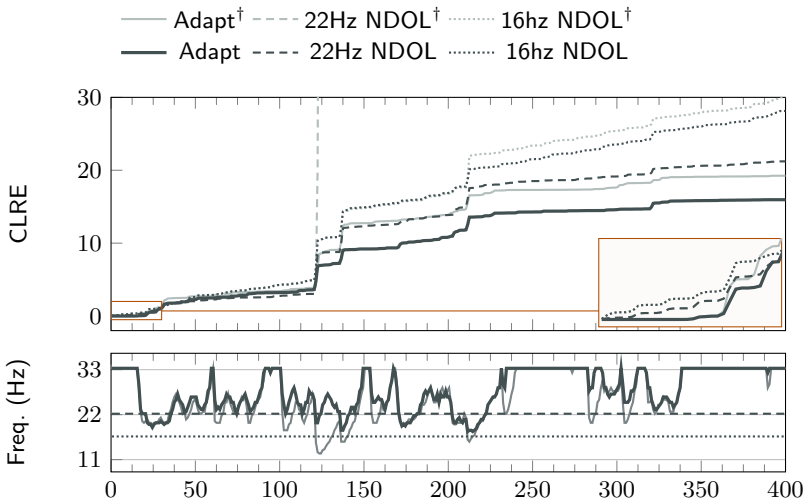


Figure 13.10 Accumulated closed loop error

the use of the adaptive controller in general, but to see if it consistently provides good performance in these scenarios.

Results from Scenario 1 are shown in Figure 13.10. The figure shows the CLRE in the upper graph, and frequency in the lower graph. Notice that the simulation is now 400 s. The box in the lower right corner of the CLRE graph shows a scaled up version of the first 30 s. This, zoomed time frame, shows how initially the adaptive controller has the best performance, as it is starting at a high frequency. As observed previously, there is a large increase in processing times at around sixteen seconds. At this point, the error of the adaptive controller distinctly increases, and the Adapt^\dagger temporarily becomes the worst performing mode. Later, the mode performs better, in relative terms, and soon surpasses both 16 Hz modes in performance. In the end, it also outperforms the 22 Hz NDOL mode.

The error of mode 22 Hz NDOL † goes straight up shortly before 120 s. At this point, the 22 Hz NDOL † client falls back on the recovery controller due to latency buildup. The mode will recover, but its error is off the chart. This mode will also run into a similar situation later in the simulation. Meanwhile, the adaptive controller often uses a frequency around 22 Hz, but must resort to lower frequencies at times. Especially, the \dagger mode, which moves far down in frequency at the point where the 22 Hz NDOL † mode takes off, as seen in the frequency graph.

These simulations create conditions that the resilient controller should be prepared for, but which are expected to be of transitory nature and not

Table 13.4 Closed loop error response (Equation (13.13)) after executing the scenarios in Table 13.1 for 400 seconds.

	Adapt	Adapt [†]	22.2 Hz NDOL	22.2 Hz NDOL [†]	16.6 Hz NDOL	16.6 Hz NDOL [†]
Scenario 1	15.96	19.25	21.22	307.87	28.14	30.00
Scenario 2	20.19	21.06	24.77	25.10	30.15	30.15

occur frequently. The adaptive modes will get to execute some sequences at the highest frequency, but there is also a lot of variation in the simulations. To see the mode switches of Scenario 1, tables with state transitions are supplied in Appendix A. The results show that the adaptive controller not only manages to avoid the backup controller under these conditions, it also manages to outperform its rivaling cases while doing so.

A simulation of Scenario 2 was also run, with results presented next to Scenario 1 in Table 13.4. Given that 22 Hz NDOL[†] comes in very close to 22 Hz NDOL for Scenario 2 in these results, we see that the problems from Figure 13.8 are remedied by the reduced frequency. For the 16 Hz modes there is no difference at all. The adaptive modes, however, provide the best scores, and the Adapt mode again has lower error than Adapt[†], although this difference is smaller compared to Scenario 1. These results show no indication of a negative effect of introducing the adaptive controller.

14

Resilient Elastic Control in the Cloud

This chapter returns to the cloud native testbeds, in order to study the frequency adaptive design in combination with real clouds. This serves two purposes. The first purpose is to observe the control system on real clouds, with the intention of validating the design. The second purpose is to demonstrate and evaluate the control system in combination with the clouds elastic properties (i.e. resource scaling). The chapter first presents implementation details for the frequency control from Chapter 13, then evaluates the result in four scenarios.

The resulting control system is referred to as a Resilient Cloud Control System (R-CCS). The R-CCS exhibits quality elastic properties, and targets an elastic infrastructure. There are four identified challenges for the R-CCS.

Resilience A R-CCS must be tolerant to non-trivial delay distributions and connectivity issues. This can be considered a form of partition tolerance, something that distinguishes the R-CCS from other control systems. Partition tolerance necessarily requires that the system remains in a recoverable state, not requiring manual intervention, even if the remote extension becomes unusually unresponsive, or is completely removed.

Performance The R-CCS should not rely on fault tolerance mechanisms that systematically penalize or degrade general performance, i.e., when the system is not subject to loss. Under nominal conditions, the control system should achieve good, nearly optimal, performance.

Costs Failed requests that happen frequently constitute a resource waste. This can incur a monetary cost and can have adverse system effects. To reduce resource waste, a design that allows the specification of a maximum loss is desired.

This chapter is based on [Skarin et al., 2021]

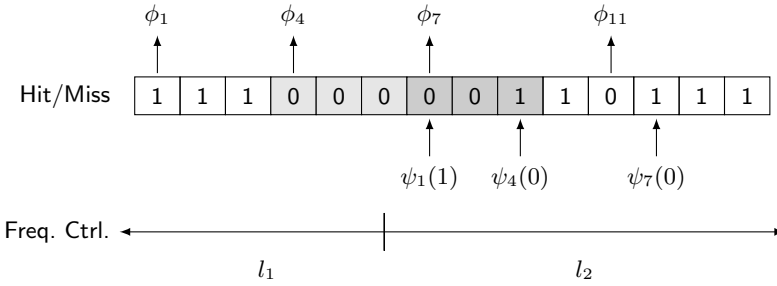


Figure 14.1 Counting hits and misses. Below: Two sample inputs to the frequency control.

Quality-of-Service The R-CCS assumes that the execution platform provides consistent quality of services most of the time, but allows arbitrary, temporary degradation. The quality of service level varies on different platforms, and can also change over time.

The previous chapter showed results from using two methods to meet these challenges. The first was execute to completion, with delay mitigation. Basic mitigation was implemented using the open loop sequence. The second method was to use frequency adaptation to increase the chance of closed loop control, and to mitigate congestion. Details of the frequency controller were left out of the previous chapter. This chapter begins by detailing the frequency controller, before moving on to the experimental setup and the results.

14.1 Frequency Controller Details

The frequency controller is implemented using a PI controller, building on similar work from [Ploplys et al., 2004; Xia and Zhao, 2007; Björkbom et al., 2010]. Specifically, parameters from Xia and Zhao form a basis for the frequency controller due to the similar range of sampling rates. The input to the controller is a loss rate, ρ , which was introduced in Figure 13.5. The output is a control period h_d , for use in the following optimizations. The request frequency follows from the control period, $f_c(t) = 1/h_d(t)$. The frequency controller has its own sampling frequency h_f .

Miss ratio

Usually, ρ is calculated from the number of missed deadlines. However, because the client execute at a relatively high rate, and delayed responses can be applied, a version was adopted that provides a measure of experienced delay. This is illustrated in Figure 14.1. The upper part shows how hits and

Table 14.1 Results of frequency adaptation implemented to track latency using the procedure in Figure 14.1 (Adapt₁ and Adapt₁[†]), and when counting missed deadlines (Adapt₂ and Adapt₂[†])

	Adapt ₁	Adapt ₁ [†]	Adapt ₂	Adapt ₂ [†]
Scenario 1	15.96	19.25	17.16	15.18
Scenario 2	20.19	21.06	22.27	22.06

misses are registered. The observed sequences are stored in a buffer which forms input to the frequency controller. l_1 and l_2 shows ranges on each side of a sampling point, where the frequency controller is updated using l_1 . l forms the miss ratio

$$l(k) = 1 - \frac{1}{L} \sum_{i=1}^L \delta(i) \quad (14.1)$$

where δ represent the buffer array and L is the buffer length. In Figure 14.1, the response ψ_1 arrives to apply the open loop action $\psi_1(1)$ but missed the closed loop action $\psi_1(0)$. The intended activation time of $\psi_1(0)$ is shown as the light gray region. The response $\psi_4(0)$ arrives within its activation time, shown in a darker gray, and counts as a hit from when it arrived. Note that there is a frequency change at ϕ_7 and the hit from $\psi_4(0)$ is extended until request ϕ_{11} . Also, consider that $\psi_4(0)$ was further delayed, but $\psi_7(0)$ arrived before its activation point (i.e. before ϕ_{11}). There will be a choice of selecting miss because $\psi_4(0)$ was delayed, or a hit because $\psi_7(0)$ was observed early. The results in the previous chapter were obtained using the latter alternative.

For reference, results from the simulations in Chapter 13 are repeated in Table 14.1 as the pair Adapt₁ and Adapt₁[†], next to the pair Adapt₂ and Adapt₂[†]. The latter are results from a simulation where the miss ration l is instead counted as missed deadlines, over the interval h_f . Adapt₂ and Adapt₂[†] often use a lower frequency than the counterpart in Adapt₁ and Adapt₁[†], which explains the relatively good response for Adapt₂[†]. Recall that the † cases can overload the service, causing queued request that accumulate delay. An already lower frequency will be less susceptible to load changes. The mode represented by Adapt₁ and Adapt₁[†] is chosen in the following experiments for two reasons. First, because the non-† cases represent the targeted behavior of the cloud service, and second, the higher request frequency should pose a greater challenge, with more load imposed on the service.

The miss ratio is used as input to an exponential moving average (EMA) filter,

$$\rho(t) = \begin{cases} 0, & t = 0 \\ \alpha l(t) + (1 - \alpha)\rho(t - h_f), & t \geq 0. \end{cases} \quad (14.2)$$

to smoothen the loss value. α is the smoothing factor of the filter, $0 < \alpha < 1$.

Table 14.2 Value offsets, and parameter μ of the lognormal distributions in Figure 14.2. σ is always one. The top row shows the start time of the distribution.

	0s	10s	40s	60s	70s
Offset/ μ	15/0.78	45/1.02	28/-1.27	28/0.32	28/-1.27

Continuous time, t , is used here to signal that the frequency controller works at its own rate. The filter is present in Xia and Zhao but it is specified differently from Equation (14.2). Because the original filter has a very limited smoothing effect, an EMA is assumed instead in the following examples. The alternative is not useful when increasing the rate of the frequency controller.

Controller

The miss ratio is used to modify the effective sampling rate of the remote controller using PI control. The design in Xia and Zhao uses PID control, where the control signal is the sampling period of the MPC, i.e.,

$$\dot{h}_c(t) = K(\dot{e}(t) + e(t)/T_i + T_d\ddot{e}(t)), \quad (14.3)$$

where dot notation marks the first and second derivatives with respect to time. $K \in \mathbb{R}^-$, $T_i \in \mathbb{R}^+$, and $T_d \in \mathbb{R}$ are the gain, the integration time constant and the derivation time constant, respectively. This expression is discretized, using backward difference approximation, with the frequency controller sampling rate T_f , to form

$$\begin{aligned} \Delta h_c(k) = & K_p(e(k) - e(k-1)) + K_i e(k) \\ & + K_d(e(k) - 2e(k-1) + e(k-2)), \end{aligned} \quad (14.4)$$

where $K_p = K$, $K_i = KT_f/T_i$, $K_d = KT_d/T_f$. Notice that K is a negative number. This is due to the definition of the error, e , as a measure of *loss*, ρ in relation to a *targeted loss*, ρ_r ,

$$e(t) = \rho_r - \rho(t). \quad (14.5)$$

When $e(t)$ is positive, the loss is low, allowing for an increased frequency. This translates to a reduced sampling period, and therefore a negative output from the controller. The control signal $h_c \in \mathbb{R}$, $h_{min} \leq h_c \leq h_{max}$, $h_{min} > 0$ is the input to the MPC control period h_d , which is rounded to the nearest integer multiple of h_q .

Four modifications are made to the design in Xia and Zhao (frequency range and loss setpoint not included): the controller gain is increased, the sampling period of the frequency controller is reduced, the derivative term is

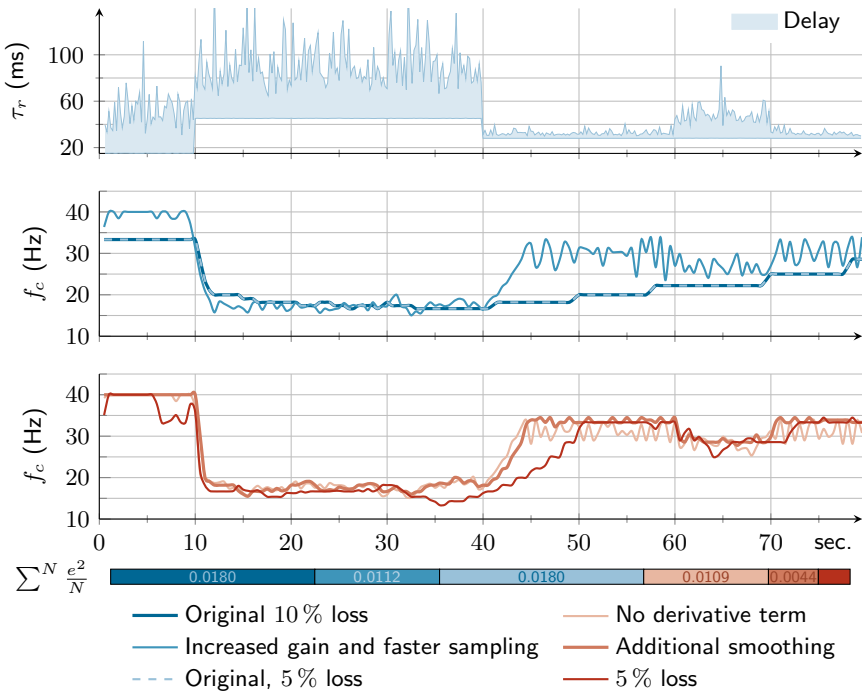


Figure 14.2 Frequency controller response to delay. A graph of the delay at the top is followed by two plots showing the controller output. The bar below shows the mean squared error. The final value, missing from the dark red box, is 0.0028.

dropped, and the smoothing parameter of the filter (Equation (14.2)) is modified. These modifications are motivated by the result shown in Figure 14.2. The figure shows six different controller responses, reacting to the delay profile in the upper plot. The delay is created from arbitrarily selected lognormal distributions, the parameters of which are listed in Table 14.2. The values in the frequency plots are averaged over half a second.

The solid, dark blue line, in the middle plot of Figure 14.2, shows the original design. The parameters of this controller are shown in Table 14.3 (although, the accepted frequency range is different). Note that the gains in the table, K_p , K_i , and K_d , are scaled by 1×10^3 . While this configuration could be used, the rise after 40 s is slow. There can also be an issue with the overload scenarios, studied in the previous chapter. When the controller is causing a high load, a sampling rate of 0.5 s can cause a substantial amount of requests to pile up, before the frequency controller reacts. From these two observations, a configuration with an increased gain and reduced sampling

Table 14.3 The original configuration from [Xia and Zhao, 2007], intermediate configuration for reference in the text, and the final configuration.
* These values are scaled by 1×10^3

	K_p^*	K_i^*	K_d^*	α	T_f	ρ_r	h_{min}	h_{max}
Original	7	6	3	0.7	0.5 s	10 %	10 ms	30 ms
Intermediate	35	1.2	3	0.38	0.1 s	10 %	-	-
Used	35	1.2	0	0.1	0.1 s	5 %	20 ms	100 ms

period is attempted.

The medium dark blue line in the middle graph shows the result. While this configuration meets the objective of a faster response, it is unsatisfactory in terms of oscillations, especially from 40 s and on. The used parameters are listed as the Intermediate configuration, on the second row of Table 14.3. Note that α has changed, in addition to the gains. This is done to retain the time constant, τ , (the reactivity) of the filter, when changing the sampling rate. The time constant is defined as the time it takes for the filter to reach 63.2% of the true value in response to a step function. It is calculated as

$$\tau = -\frac{T_f}{\ln(1 - \alpha)}, \quad (14.6)$$

and has a value of ≈ 0.42 in the original design. If α is not changed, the time constant reduces to ≈ 0.08 . This small time constant provides very limited smoothing and is not useful.

The oscillations of the faster controller are dampened by removing the derivative term, as seen in the bright red line of the bottom graph. Primarily however, the issue is due to insufficient smoothing, which is why α is adjusted to 0.1, to put less emphasis on recent values. The response is shown in the medium dark red line of the bottom graph. At this point, the controller has a much faster rise and a smooth response. With sufficient damping, the derivative could be reintroduced. It will, for instance, slightly decrease the steep fall at 10 s. To keep things simple, the derivative term is not used.

To arrive at the final configuration, as used in the previous chapter, the loss setpoint is reduced from 10% to 5%. The new setpoint causes 'dips' in the frequency and a slower rise time, as seen from the dark red line in Figure 14.2. Notice that the changed setpoint has no effect on the original configuration, as seen in the dashed blue line in the middle plot. This is an arbitrarily chosen value, to test a more demanding configuration. Notably, however, further lowering the setpoint will create a conservative controller, recreating the slow rise time of the original. This is starting to show in the 5% case, while a loss of 7% (not shown in the graph) would place the response much closer to the response at 10%.

14.2 Experimental setup

The experiments in this chapter are aimed at demonstrating the feasibility of the proposed R-CCS using realistic scenarios. The objectives of the experiments are to:

- 1) *Validate the frequency adaptation controller.*
- 2) Demonstrate the *individual and combined effectiveness of the methods* initially presented in Chapter 13, i.e. frequency adaptation, predictive control and recovery control.
- 3) Demonstrate that a R-CCS addresses the following challenges:
 - a) is able to cope with *performance transients in the infrastructure.*
 - b) is able to prevent *resource starvation.*
 - c) is able to successfully *transition between heterogeneous cloud platforms.*

The experiments were conducted using a set of heterogeneous cloud deployments and a real-time simulation of the reference plant. The control system is deployed on four cloud platforms. The evaluation in these experiments is aimed at the relative setpoint tracking performance of different configurations. As was seen in Chapter 13, the goal is to have a robust frequency control and avoid using the backup controller. For the purpose of these experiments, it is enough to verify this through visual examination of the results. To exercise the proposed R-CCS, all experiments are subjected to noise and disturbances throughout the duration of the experiments. Gaussian noise, $\mu = 0, \sigma = 3$, is added to the control signal u . Additionally, the position of the ball on the beam is altered between $(-0.5, 0.5)$ every 10 s. The noisy control signal provides a complementary benchmark to the disturbance used in Chapter 13. Unless otherwise stated, all experiments run for 120 seconds.

Implementation

The optimization services are implemented in Python, using CVXOPT as the optimization framework. The implementation is accessed over a persistent HTTP connection. Multiple parallel connections can be made, thereby enabling execute to completion. The async framework of Python is used for this purpose. A POST request containing the predicted state of the plant, and other parameters necessary to construct Equation (13.8), will yield a response containing u^* and the predicted states \hat{x}^* . The latter is not used in these experiments.

To the extent possible, requests are sent periodically with low jitter. For this purpose, high priority real-time modes of the Linux kernel were used. The deviations from the intended actuation times were also measured, and

determined small in relation to the control periods. This is important, not for the plant, which is simulated, but for a correct request sequence towards the remote services.

Cloud platforms The four cloud infrastructures are heterogeneous in capacity, hosting platform and network proximity. The infrastructures are:

K8S: A Kubernetes cluster of seven bare-metal nodes located close to the client. The controller service is exposed using an nginx ingress controller. It has a measured median and 95th quantile RTTs of 11.71 ms and 12.82 ms, respectively. This is the baseline deployment.

ERDC: The previously used Openstack-based research-focused DC located 1.3 km (0.6 miles) from the client. The controller service is hosted on four VMs and ingress traffic is routed through an instance of HAProxy. Measured median and 95th quantile RTTs are 24.24 ms and 26.46 ms, respectively.

Central & North: AWS Lambda functions with 1024 MB RAM and no throttling. They are hosted in Frankfurt, Germany (eu-central-1) and Stockholm, Sweden (eu-north-1). The measured median and 95th quantile RTTs are 37.55 ms and 52.62 ms for eu-central-1, and 180.06 ms and 218.57 ms for eu-north-1. The controllers are exposed using AWS API Gateway.

Experiments

The following experiments were defined to demonstrate and validate the R-CCS.

Single controller These experiments provides a first insight into how capable the platform is in hosting a single controller. A pure MPC without mitigating features is evaluated, and three configurations from Chapter 13: the SPSC, the NDOL, and the adaptive controller, now referred to as the R-CCS.

Infrastructure disruptions and transients This experiment is intended to exercise and validate the methods using a transient infrastructure, allowing for all system dynamics to interact constructively. The transient behavior is representative of events such as varying network load and consequences of infrastructure resource management policies. Building on the results from the single controllers, this evaluates a R-CCS's ability to cope with disruptions in the infrastructure. The experiments use the K8S cluster. The cluster is subject to a periodic delay using the Chaos Mesh¹. Specifically, the NetworkChaos module was used to subject one Kubernetes pod (a

¹<https://chaos-mesh.org>

service instance) to a delay with a mean of 100 ms, correlation of 25, and a jitter of 15 ms, for 30 s every 1 m.

Resource starvation The proposed R-CCS is quality elastic, having the potential to suppress resource starvation caused by contention between multiple cloud tenants (multi-tenancy), without fully compromising the performance of the tenants. This aspect is relevant in terms of resource usage, but also in terms of resiliency, as it may take the cloud some time to respond to an increase in the workload. These experiments, evaluate the impact of successively applying more tenants to a resource constrained instance of a remote controller service. Each tenant is a new MPC offloading client, in control of another ball and beam plant. Three plants are used, admitted 20 s apart, all using a controller service deployed in K8S. At the start, the controller deployment is hosted on one Kubernetes pod. After all three plants have been admitted into the cluster, the deployment is scaled to three pods, at $t = 60$ s. In addition to the basic criteria of success, the success in this experiment is also determined by the ability to share resources and maintain stability.

Transition between cloud platforms Although the proposed R-CCS may be able to cope with a transient cloud infrastructure, any controller has an upper RTT limit, beyond which the network controller is not useful, or worse, unstable. If, at any point in time, that bound is violated, or another cloud can offer better performance in terms of control frequency, an R-CCS shall be able to seamlessly switch to that cloud. This returns us to the migration experiments in Chapter 7, but now implementing the feature using a service architecture. In the experiment, a controller deployment is randomly selected (other than the current) among the cloud infrastructures detailed in Section 14.2, every 20 s. The criteria of success is the R-CCS's ability to successfully transition between cloud deployments and timely adapt the control frequency of the process to that deployment.

14.3 Results

Single controller

Figure 14.3 shows the over-laid time-series of the outcome from the four configurations, the MPC, the SPSC, the NDOL, and the R-CCS. As usual, the top plot shows the critical position of the ball, with a square-wave setpoint. The middle plot shows the frequency, which is at 33 Hz for all configurations except the R-CCS. The bottom plot shows load, defined as the ratio of execution time versus the control period. For visibility, the load is only shown for NDOL and the R-CCS.

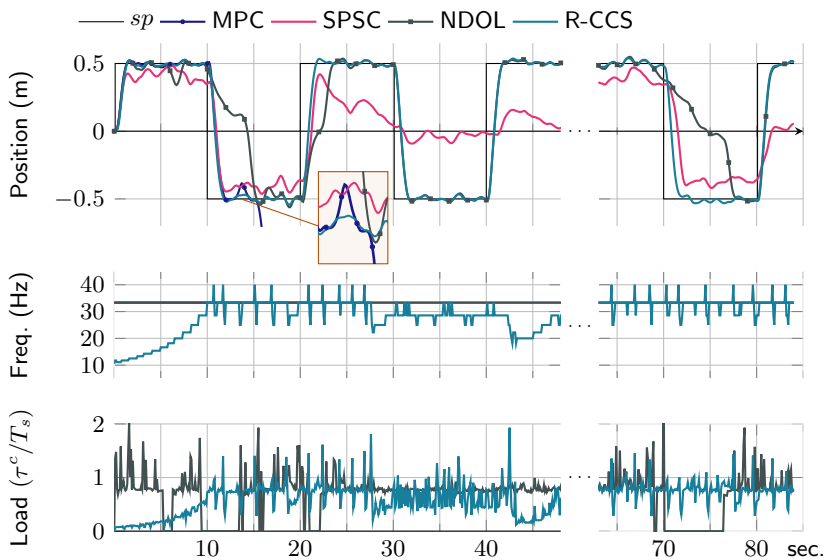


Figure 14.3 Evaluation of the merits of individual remedies and combinations of remedies, including the proposed R-CCS. sp is the position setpoint.

It is not long until the MPC fails, at around 15 s. The trajectory of the MPC falls off the chart at the second setpoint change, at the position marked by a zoomed region. This is due to successive losses, no ability to adapt, and no fall back on a local controller. This shows that the predictive controller is not able to stabilize the plant under the conditions of the experiment. Also observe that just before this failure happens, there is a visible bump in the trajectory of the MPC. This shows how the applied disturbance can affect the system when there is no control signal to handle it.

Adding the local recovery controller, yields the SPSC configuration. The results are similar to the experiences from the simulations in Chapter 13. Initially, the SPSC has some degree of performance, but does not reach the setpoint. After some time it completely falls back on the backup controller. It later recovers, and as seen in the 70 s to 80 seconds on the right.

Introduction of an acceptable control lag, in configuration NDOL, provides an improved response. This configuration is able to keep the ball on the beam and track the setpoint most of the time. Significant effects on performance, due to losses beyond the lag, are observed in relation to setpoint changes, in the ranges $t \in (10, 15)$, $t \in (21, 24)$, and $t \in (69, 78)$. Effects of delay are also visible for NDOL at 6 s and 16 s, where the offset to the setpoint becomes significant and clearly different from the R-CCS.

The load plot shows that the problems experienced by the NDOL are due to processing load. During the time frame 0 s to 25 s, and in the plots on the right side, a processing load above one is occurring frequently. This leads to extensive delays and reduced control performance. Lower load is experienced from 25 s to 50 s, and here the controller works well.

Sequences where the load is zero, are due to timeouts of the control requests. This could happen due to network delay but also because the remote service is overloaded by optimizations. When a request is sent, the maximum lag, σ , is set as a timeout on the connection. When this timeout is reached, the connection is closed and no load is registered for the requests. Because persistent connections are used, the cloud service learns about the timeout when the connection is closed and shuts down the optimization. This helps lowering the load, compared to the simulations in Chapter 13 where there was no deadline. The NDOL mode repeatedly recovers, and can continue to track the setpoint.

The frequency adaptive configuration gives the best result. It consistently shows a quick and smooth response to setpoint changes and attenuates disturbances well. The load plot also shows that the R-CCS is experiencing loads above one, but frequent rate switching allows it to perform adequately. In the frequency plot of Figure 14.3, the control frequency f_c starts at 10 Hz and the R-CCS successfully adapts the control frequency within 10 seconds. It settles at a median control frequency of 33 Hz, equal to the other configurations. Later, $t \in (28, 50)$, response times increase, and the R-CCS adapts to a lower control frequency. When looking at these results it is important to note that the conditions are not equal between the various configurations. This can be seen in the range 28 s to 50 s, where the load situation has clearly improved for NDOL. This is contrary the situation of the R-CCS. The large variation in the load after the first frequency reduction makes it reasonable to conclude that the load on the cluster has changed. Meanwhile, the reduction at 45 s may be due to a temporary increase in the network or admission time. We can conclude from this, that the R-CCS design that was simulated in Chapter 13, is also working efficiently in a real setup.

The oscillations in the frequency are due variations in the computation time. Spikes are visible both in the frequency plot and the load plot, and they are more pronounced at high frequency than low. This indicates a somewhat aggressive controller, but is also a consequence of disturbances acting on the system, while the controller is acting close to constraints.

Infrastructure disruptions and transients

Figure 14.4 shows the over-laid time-series of the outcome from the NDOL and the R-CCS when subjecting the hosting cloud infrastructure network to an injected delay. The change in delay is seen from the RTT in the bottom

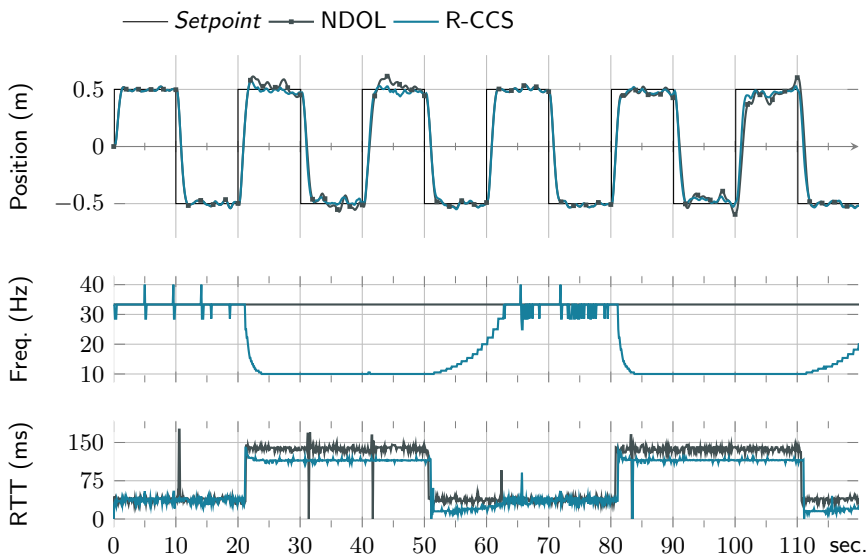


Figure 14.4 Time series when applying a delay disturbance to the cloud deployment, comparing fixed and adaptive frequency, as detailed in Section 14.3.

graph of the figure. The mean delay disturbance of 100 ms is within the maximum allowed control latency, σ , of 210 ms. Therefore neither configuration will intentionally be pushed beyond the point where it will not receive any responses. Position and frequency are shown as before.

During the two periods of large network delay, the R-CCS reduces the control frequency to match the experienced request RTT, pushing the frequency to the very bottom of its range. This also affects the computation time, and therefore the RTT of the R-CCS becomes lower than the NDOL. The net effect is that the NDOL has to compensate for a larger delay than the R-CCS, and clearly suffers a worse response. This experiment shows the effect of the noise that is added in the simulation. After reaching the setpoint, the NDOL has problems attenuating this disturbance when the delay is increased. This also affects the R-CCS but to a lower extent.

In this experiment, the R-CCS is compensating for an independent delay imposed on all requests. This could also be handled by latency compensation, without modifying the frequency. However, it shows the mitigating properties of the R-CCS, and while the transmission delay is dominant, there is a double effect in the extended dead-time and lowering the system load. The latter provides a shorter response time and if the delay was dependent on the request frequency of the controller, this could have further effect. The

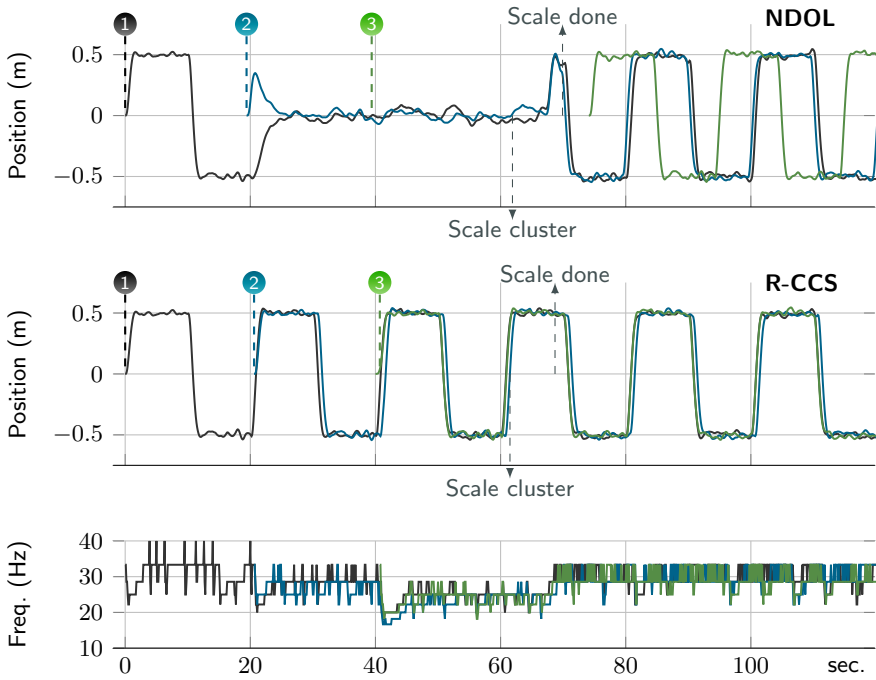


Figure 14.5 Time series of outcome of resource starvation, comparing fixed and adaptive frequency, as detailed in Section 14.2.

controller at the same time creates more room for processing load to occur, which makes the controller less effected by load variation while the transmission delays are high.

There are some other things to note here as well. It should be reiterated that the execute to completion strategy can be enhanced with a delay compensation, which will improve the response of NDOL. This would not remove the necessity to use an R-CCS, but in a scenario such as this one, frequency reduction must be contrasted with compensation. Another thing to note is that, over the course of the experiment, the accumulated error of the R-CCS compared to NDOL is 11% lower, while at the same time it uses 51% less execution time. A lower error and lower execution time translate to a higher yield per resource, lower resource usage, and potentially reduced costs. This must also be weighted in, if considering delay compensation to raise the frequency of the controller.

Resource starvation

Two experiments are now performed to further explore the property of reducing the load on the hosting infrastructure. With the control service hosted in K8S, Figure 14.5 shows responses for plants 1,2, and 3, successively admitted 20s apart. The first chart shows the outcome when using the NDOL.

When one plant is using the cluster, from 0 s to 20 s, the cloud-deployment is able to accommodate the MPC controller, executing at 33 Hz. At 20 s, a second plant is admitted, and at 40 s, a third. There is a visible attempt by the second plant to move towards the setpoint, but it quickly has to revert to the backup controller, bringing the position back to zero. This also brings the first controller to zero, showing that the cluster has already become overloaded. The third controller does not even start, as seen by the offset in the trajectory from the third client. The client is implemented to not start its closed loop process until the first control response is received from the cloud. Because the cluster is overloaded, the third client receives no responses and its start is delayed.

At around 60 s, the cluster scales from one to three worker pods. When scaling is done, all three plants can be accommodated, and subsequently they leave the 'go-back-home'-mode and resume to successfully track the setpoint.

When employing adaptive control frequency, i.e., using the R-CCS, all three plants successfully track the setpoint, with little or no error. Independently but in unison, the plants reduce their control frequency until they reach the tolerated level of loss, for each successive plant admission. An unanticipated, positive, side effect is that the entry of a new plant is not at all observable in the responses, at least not from a visual examination of the position graph. Meanwhile, the entry of a new plant very abruptly affected the outcome when using NDOL.

The admittance of new plants is visible in the frequency graph at the bottom. Notice the difference to the previous experiment. The external and persistent disturbance in Figure 14.4 causes a fast frequency reduction, and later a slow ascent back to the high frequency. This was also seen frequently in simulation. Here we see how all plants quickly reduce frequency, compensating for each other, then progress back to a useful frequency. While there is an overshoot to a slightly lower frequency, the response is fast and the clients never have to resort to the lowest frequencies.

At 60 s, the deployment again scales to three pods. As shown in the bottom chart in Figure 14.5, each plant independently and opportunistically increase their control frequency. The control frequency, after the deployment has scaled, is lower than with one pod and one plant. In theory, the three pods can handle one client each, and they can all execute with the frequency initially observed for the single controller. This lower performance can be caused by increased network activity, admission control, insufficiently pow-

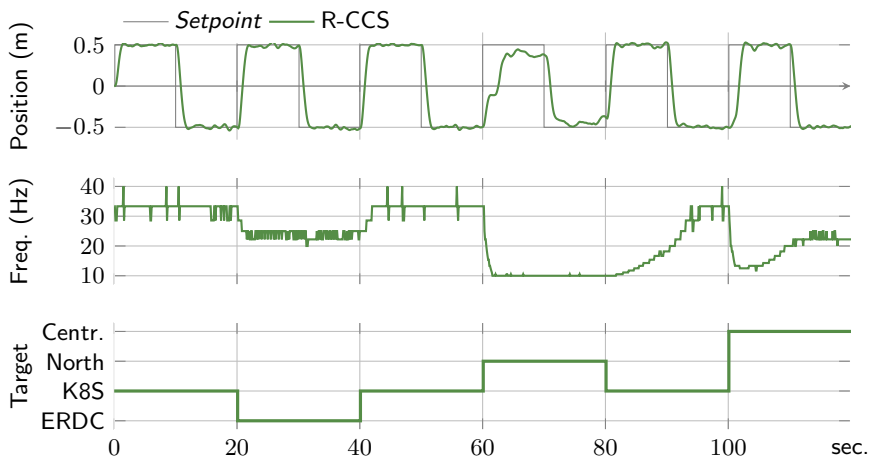


Figure 14.6 Time series of outcome of one system transitioning between set of heterogeneous cloud platform.

erful worker nodes, or unevenly distributed load. Ideal performance may be hard to achieve without a lot of over provisioning, especially in more realistic scenarios, with more variation in the clients and control problems. It is a useful property, that the infrastructure can be probed and configurations tuned while systems remain on-line.

Transition between cloud platforms

At this point, there is no reason to believe that the R-CCS would be incapable of transitioning between deployments. In Chapter 7, this was accomplished by deploying a runtime in virtual machines on two clouds, in a local resource-constrained device, and on an edge resource at the base station. A resilient design was not used in Chapter 7, and the offloaded structure was not used. We have yet to demonstrate the potential of the service deployment, i.e., with offloading using FaaS. The device now hosts the client, the edge device is replaced by the local K8S cloud, the small scale data center remains present as ERDC, and large scale services are offered by the North and Central AWS in Europe. Communication is over HTTP, while the runtimes in Chapter 7 would use TCP, or some other protocol, chosen by the platform. Cloud native frameworks that provide scaling, introspection, and common software development tools are used. In each of the locations, an idle, zero cost function is stored and waiting for requests. The only thing the client needs in order to execute at another location, is to modify the address of the request. We now perform the experiment, similar to the Calvin migration, knowing that if only the transitions work well, the resilient, offloaded controller will not

fail, and should adapt its performance to the new deployment.

One client and one plant is used. The client randomly switches between the four cloud deployments detailed in Section 14.2. As demonstrated in Chapter 8, the deployments have different properties, but the client makes no difference between them. Nevertheless, the R-CCS is able to seamlessly switch between the locations, and expediently adapt to a control frequency appropriate to that deployment. In Figure 14.6 the upper chart shows the position of the ball, below, the target cloud, followed by the controller frequency.

The deployment North has the highest prior measured RTT and the most variations in processing time, and is therefore the most challenging. The deployment K8S (from 0s to 20s and 40s to 60s) has the lowest measured RTT, and this is also where the highest frequency is reached. Between 20s to 40s the municipal data center, ERDC, is used. There are many, small, frequency variations observed during this time, but this has no substantial effect on the closed loop response. At 60s, the deployment changes to a distant, large scale data center service. Because of an increase in delay, the control frequency is reduced to a minimum, but this compensation is not enough. The client struggles to track the setpoint as the backup controller repeatedly forces the plant towards the origin. At 80s, when the service goes back to K8S, tracking performance is immediately regained, and fifteen seconds later, the high frequency mode is achieved.

The frequency drops again when the deployment is changed to another large scale service. Clearly, the frequency is initially underestimated, as the frequency controller reacts to a sudden large loss, but over the course of a few seconds, the controller achieves a reasonable rate. Notice that Central is a Lambda service, while in Chapter 7 a VM was used towards the same cloud. The settled frequency is slightly higher than what was used on the central cloud in Chapter 7, and now the implementation is also robust and using higher level APIs. The sequence in Figure 14.6 is short, but indicates that the service will work in practice. Another thing to note is that Central and North are using a more costly configuration than what has been used towards Lambda previously in the thesis. This does not provide a positive observation of results on North but is one reason for the positive result for Central. This shows the flexibility of the cloud. The upgrade only requires changing a value in a web interface, and this can be done independently of the client, while it is executing.

14.4 Conclusions

This chapter and Chapter 13 have presented a resilient cloud controller design, through an offloading architecture and frequency control. The output

of a loss ratio-based smoothing filter and PI control is used to manage the client request rate. The result implements constrained MPC using the cloud. To ensure reliability, the client implements a backup controller and a 'go-back-home' mode, in case the R-CCS fails its objective. Simulation and experiments on clouds, show how the Resilient Cloud Control System is able to mitigate delay, allow a cluster to admit new controllers while scaling to meet the new demand, and change its deployment on-line. These positive effects are achieved, while at the same time keeping the number of unused, wasteful, requests low. Further work on the R-CCS can introduce other means of altering the controller load on-line, e.g., built-in delay compensation, statistical methods for setting the frequency, and improved automation through machine learning.

Part V

Conclusions

15

Conclusions and Future Work

Based on the prospect that many control systems will be connected to cloud services, this thesis has explored cloud aspects from the perspective of control engineering. In contrast to many related works, the method has been to focus on redesigning control systems for using utility computing, in order to explore this potential and the uncertainty of using clouds to implement critical control systems. To that end, it has examined generic principles, elementary properties, and constructed prototypes to examine cloud native platforms. The goals of the individual works have been to answer questions on the state of current technology, demonstrate properties of cloud control systems, and develop principles for resilient cloud control. The first part of the thesis introduced the cloud control challenge and offloading. In controller offloading, a client requests services from the cloud to replace its control function. In the thesis, this is used to execute optimization problems, requested on an as-needed basis by the client.

The second part of the thesis studied performance using two testbeds. The first testbed implemented a controller on a PaaS, deployed in a distributed cloud. The platform traversed wireless and public networks, and supported software migration, allowing placement of the controller on a resource-constrained device, an edge cloud, or in distant data centers, and relocating it dynamically while controlling the plant. This demonstrated that such a platform is able to support the reference system, but also showed problems related to both networking and computation delay. Good performance was only achieved at the edge, communicating over state of the art wireless communication using the massive MIMO research testbed LuMaMi.

In the second testbed, the control system was implemented to be compatible with FaaS. This was used to measure response times in the cloud stack and processing time variations on different platforms. Results indicated that this design can also be used to control the reference plant, which

was further verified using two versions of an offloaded client. The second client achieved limited gains in terms of performance, but in return provided good resource utilization and no constraint violations. Further, the measurements illustrated the overhead of using cloud native services and high level interfacing, and the impact of distance. A cloud within a few kilometers was compared to one hundreds of kilometers away, but within nation borders. This showed the theoretical limit on achievable response time, and the impact of communication layers and intermediate networks. In this specific scenario, an offloaded task will have to execute several times faster in the hyperscale data center, in order to compensate for access time overhead.

Control strategies were investigated in part three. First, parallelism was used to implement a variable horizon controller. With every sample, the client sends several requests to the cloud, and later chooses what action to apply based on the responses it has received on time. Each optimization implements an ordinary MPC with nominal stability guarantees. The client must handle the event of not receiving responses, and the horizon can change between control actions. The evaluation shows that the controller is executing reliably and that it is able to handle request loss. The use of an edge cloud was also considered. This illustrated the use of two different locations in a distributed cloud, to improve the response while maintaining the basic structure.

The second strategy introduced rate switching. In this case, the client requests support from the cloud to execute an MPC at higher rate than what is possible locally. This creates another switching system, with alternatives for when to execute the low frequency controller and how to handle the event when neither of the two optimizing controllers provide a response. Several modes were examined and simulated to show relative performance improvements. It was also shown that the configurations can experience problematic maximum constraint violations.

The final strategy expanded the control problem to ensure constraint satisfaction in the event of request loss. This was done by introducing recovery paths, explicitly verified in the remote MPC. This comes at a computational cost but provides flexibility, separation of concerns, and a reliable way to override a device controller with an advanced alternative executing in the cloud. The evaluation examined effects on performance with various configurations of the recovery.

Part four took on a perspective of cloud control that is important independently of how the controller is implemented: sharing resources and scaling. Before demonstrating scale-out on the cloud testbed, a resilient controller was developed using predictive control, control frequency adaptation, and a *go-back-home* mode. Simulations based on previously recorded data were used to assess the strategy and show that it has a positive effect on a single control system. The strategy was then applied when executing on the testbed, first applying artificial loads, and then in a scenario with three controllers

and cluster scaling. The results showed good and reliable performance. The adaptive controller reached higher frequencies than what was previously configured from observation.

On-demand computing and elasticity, are intrinsic properties of clouds, and something that should be taken into account and utilized also in the design and implementation of control systems. One way to approach cloud native control, is to focus on a resilient and performance-enhancing design. Communication networks and cloud services will improve to support real-time systems, but this does not exclude the potential benefits from high average performance. With cloud, these properties may co-exist and a challenge is to create systems that can put this to good use. The thesis has taken initial steps towards such architectures, and shown that they can be applied in practice.

15.1 Future work

To introduce offloading and resilient control in engineering, the concepts must be evaluated in a larger context and applied in practice to other use cases. Preferably, this involves scenarios with several resilient controllers that exist in a larger software framework. This will help to investigate how well best practices scale and naturally include a larger part of cloud control, by introduction of related technology such as plug-and-play support, collaborating systems, and load balancing strategies. There are also many things that can be done to directly extend parts of the thesis, as listed in the following.

Performance prediction and guarantees

Theory from networked control, switched systems and robust control should be introduced to provide formal stability and constraint guarantees. In addition, on-line evaluation could predict the outcome from updated configurations of the environment, or the controller, to support automatic scaling. Introducing monetary costs in the evaluation allows systems to execute cost-effectively and with cost constraints.

Reference control problem

Future works should extend the control system to a more complex structure, such as centralizing a distributed control problem or using non-linear plant models. This can further motivate the use of offloading but also provide other benefits of using the cloud, such as executing the problem where information is gathered. Additional information can be incorporated into the remote controller, taking additional sensors or other, dependent, systems into account, when calculating the control action. This will create research in several directions, for instance: plug-and-play systems, resiliency in more

complex scenarios, best practices and limits to achievable gains, architectures that can collaborate and also use offloading, etc.

Cloud and control bench-marking

The experiments in this thesis have been formed by the perspective of distributed clouds, deploy anywhere and IoT, with focus on the architectures. Experiments have mostly used what can be considered as average performing, general purpose hardware and virtual machines. From individual providers, there are often many alternatives to chose from, designed specifically for various purposes. For instance, machines that carry a low cost but have burst performance, machines that are good at I/O operations, and machines that provide GPU acceleration. The study in [Leitner and Cito, 2016] extensively looks at virtual machines, and [Pelle et al., 2019] provides an example also looking into storage and event triggering. Similar studies, with a resilient and quality elastic client, would be useful. This can be used to verify the function and performance of elastic control designs, and domain specific benchmarks will help in selecting and evolving clouds to support these applications.

On-line statistics

Obtaining statistics on the execution environment, and using them on-line, is an interesting research direction that did not make it into this thesis. A direct extension would be to replace the PI frequency control with a different adaptation. Such an alternative can use statistics on processing times, execution times, network delay etc. This leads to data driven methods and machine learning.

Latency compensation

Latency can be further compensated in the resilient controller by applying techniques from Part III, and related theory from robust control and network control. In a practical extension, a local client might work towards the nominal path (i.e. the error $x_e = x_m - x$), and fall back to recovery if the error becomes too large or the setpoint changes. Utility computing can be supportive, evaluating several potential outcomes in the cloud, such as illustrated by the variable horizons and edge cloud in Chapter 10. Another extension is to apply a generic compensation for missing control actions that result from a delayed response and the use of open loop. Such strategies should be compared to alternatives that compensate for a certain delay profile in the controller design.

Dependent and independent delay

The resilient controller that was used in Chapter 14, will reduce its frequency even if the experienced delay is static. When the reduced frequency has lim-

ited impact on the experienced delay, it is more reasonable to work with latency compensation. Future work might combine these methods.

Update rate

The MPC controller does not have to be reevaluated for every control signal. The MPC problem can be constructed to provide a higher rate of control signals than the rate of update, i.e., control is executed in open-loop between updates by design. An update rate that is modified independent of the control frequency can serve as an alternative and extension to the frequency control in Chapter 14.

Bibliography

- Abdelzaher, T., Y. Hao, K. Jayarajah, A. Misra, P. Skarin, S. Yao, D. Weerakoon, and K.-E. Årzén (2020). “Five challenges in cloud-enabled intelligence and control”. *ACM Transactions on Internet Technology (TOIT)* **20**:1. ISSN: 1533-5399.
- Ahmad, R. W., A. Gani, S. H. A. Hamid, M. Shiraz, F. Xia, and S. A. Madani (2015). “Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues”. *The Journal of Supercomputing* **71**:7, pp. 2473–2515.
- Ahn, Y. woon and A. M. K. Cheng (2015). “MIRRA: Rule-based resource management for heterogeneous real-time applications running in cloud computing infrastructures”. In: *Presented at the Int. Workshop on Feedback Computing*. Seattle, WA, USA.
- Alessio, A. and A. Bemporad (2007). “Decentralized model predictive control of constrained linear systems”. In: *European Control Conference (ECC)*. IEEE. Kos, Greece, pp. 2813–2818.
- Alexandru, A. B., M. Morari, and G. J. Pappas (2018). “Cloud-based MPC with encrypted data”. In: *IEEE Conference on Decision and Control (CDC)*. Miami, FL, USA, pp. 5014–5019.
- Andersen, M., J. Dahl, and L. Vandenberghe (2021). *CVXOPT*. URL: <http://cvxopt.org/> (visited on 2021-07-12).
- Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2010). “A view of cloud computing”. *Commun. ACM* **53**:4, pp. 50–58. ISSN: 0001-0782.
- Årzén, K.-E. (1999). “A simple event-based PID controller”. *IFAC Proceedings Volumes* **32**:2. 14th IFAC World Congress, Beijing, China, pp. 8687–8692. ISSN: 1474-6670.
- Åström, K. J. and P. R. Kumar (2013). “Control: a perspective”. *Automatica* **50**:1, pp. 3–43.

- Åström, K. J. and B. Wittenmark (2011). *Computer-Controlled Systems: Theory and Design, 3rd Edition*. Dover Publications, Inc.
- Atzori, L., A. Iera, and G. Morabito (2010). “The internet of things: A survey”. *Computer Networks* **54**:15, pp. 2787–2805. ISSN: 1389-1286.
- Bak, S., D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha (2009). “The system-level simplex architecture for improved real-time embedded system safety”. In: *15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. San Francisco, CA, USA.
- Basiri, A., N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal (2016). “Chaos engineering”. *IEEE Software* **33**:3.
- Bellman, R. (1957). *Dynamic Programming*. 1st ed. Princeton University Press, Princeton, NJ, USA.
- Bemporad, A., M. Morari, V. Dua, and E. N. Pistikopoulos (2002). “The explicit linear quadratic regulator for constrained systems”. *Automatica* **38**:1, pp. 3–20. ISSN: 0005-1098.
- Bernat, G., A. Colin, and S. M. Petters (2002). “WCET analysis of probabilistic hard real-time systems”. In: *23rd IEEE Real-Time Systems Symposium (RTSS)*. Austin, TX, USA, pp. 279–288.
- Bernstein, D. (2014). “Containers and cloud: from LXC to Docker to Kubernetes”. *IEEE Cloud Computing* **1**:3, pp. 81–84.
- Björkbom, M. et al. (2010). *Wireless control system simulation and network adaptive control*. PhD thesis. Helsinki University of Technology Control Engineering.
- Blanchini, F. (1999). “Set invariance in control”. *Automatica* **35**:11, pp. 1747–1767. ISSN: 0005-1098.
- Bo Lincoln, B. B. (2000). “Optimal control over networks with long random delays”. In: *Proceedings CD of the Fourteenth International Symposium on Mathematical Theory of Networks and Systems*. Laboratoire de Théorie des Systèmes (LTS), University of Perpignan.
- Boberg, C., M. Svensson, and B. Kovács (2018). “Distributed cloud – a key enabler of automotive and industry 4.0 use cases”. *Ericsson Technology Review*.
- Buttazzo, G., M. Velasco, and P. Marti (2007). “Quality-of-control management in overloaded real-time systems”. *IEEE Transactions on Computers* **56**:2, pp. 253–266.
- Camponogara, E., D. Jia, B. H. Krogh, and S. Talukdar (2002). “Distributed model predictive control”. *IEEE Control Systems* **22**:1, pp. 44–52.
- Cervin, A. (2019). “Jittertime 1.0 reference manual”. *Department of Automatic Control, Lund University, Sweden, Tech. Rep. TFRT-7658*.

- Cervin, A., D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén (2003). “How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime”. *IEEE Control Systems Magazine* **23**:3, pp. 16–30.
- Chen, W.-H., D. J. Ballance, and J. O’Reilly (2000). “Model predictive control of nonlinear systems: computational burden and stability”. *IEE Proceedings - Control Theory and Applications* **147**:4, pp. 387–394.
- Christofides, P. D., R. Scattolini, D. M. de la Pena, and J. Liu (2013). “Distributed model predictive control: a tutorial review and future research directions”. *Computers & Chemical Engineering* **51**, pp. 21–41.
- CNCF (2021a). *Cloud native computing foundation charter*. Ed. by C. N. C. Foundation. URL: <https://github.com/cncf/foundation/blob/master/charter.md> (visited on 2021-06-17).
- CNCF (2021b). *Cloud native landscape*. Ed. by C. N. C. Foundation. URL: <https://landscape.cncf.io> (visited on 2021-06-17).
- Cortes, P., J. Rodriguez, C. Silva, and A. Flores (2012). “Delay compensation in model predictive current control of a three-phase inverter”. *IEEE Transactions on Industrial Electronics* **59**:2, pp. 1323–1325. ISSN: 0278-0046.
- Department of Automatic Control, Lund University (2019). *TrueTime 2.0*. URL: <https://www.control.lth.se/research/tools-and-software/truetime/> (visited on 2021-10-07).
- Dorf, R. C. and R. H. Bishop (2011). *Modern control systems*. Pearson.
- Dragoni, N., S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina (2017). “Microservices: yesterday, today, and tomorrow”. *Present and Ulterior Software Engineering*, pp. 195–216.
- Ericsson (2017). *White paper: 5G systems - Enabling the Transformation of Industry and Society*. Tech. rep. Ericsson. URL: <https://www.ericsson.com/assets/local/publications/white-papers/wp-5g-systems.pdf> (visited on 2021-10-09).
- Esen, H., M. Adachi, D. Bernardini, A. Bemporad, D. Rost, and J. Knodel (2015). “Control as a service (CaaS) cloud-based software architecture for automotive control applications”. In: *ACM Int. Workshop on the Swarm at the Edge of the Cloud*. Seattle, WA, USA.
- Feldbaum, A. (1965). *Optimal Control Systems*. Mathematics in Science and Engineering. Academic Press.
- Findeisen, R. and F. Allgöwer (2004). “Computational delay in nonlinear model predictive control”. *IFAC Proceedings Volumes* **37**:1. 7th International Symposium on Advanced Control of Chemical Processes (AD-CHEM 2003), Hong-Kong, pp. 427–432. ISSN: 1474-6670.

- Fontanelli, D., L. Palopoli, and L. Abeni (2013). “The continuous stream model of computation for real-time control”. In: *IEEE 34th Real-Time Systems Symposium*. Vancouver, BC, Canada, pp. 150–159.
- Fox, G. C., V. Ishakian, V. Muthusamy, and A. Slominski (2017). *Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research*. Tech. rep. First International Workshop on Serverless Computing (WoSC), Atlanta, GA, USA. arXiv preprint arXiv:1708.08028.
- Giang, N. K., M. Blackstock, R. Lea, and V. C. M. Leung (2015). “Developing IoT applications in the fog: a distributed dataflow approach”. In: *5th International Conference on the Internet of Things (IOT)*. IEEE. Seoul, Korea (South), pp. 155–162.
- Giselsson, P. (2015). *Qpgen*. URL: <https://www.control.lth.se/fileadmin/control/Research/Tools/qpqgen/index.html> (visited on 2021-07-12).
- Givehchi, O., J. Imtiaz, H. Trsek, and J. Jasperneite (2014). “Control-as-a-service from the cloud: a case study for using virtualized PLCs”. In: *10th IEEE Workshop on Factory Communication Systems (WFCS)*. Toulouse, France, pp. 1–4.
- Godfrey, J., C. Bernard, and N. Miller (2016). *State of the App Economy, 4th Edition*. ACT: The App Association.
- Google (2021). *Google Trends*. Ed. by Alphabet Inc. URL: <https://trends.google.com/trends/explore?date=2005-02-18%202021-03-18&q=internet%20of%20things> (visited on 2021-03-18).
- Gupta, H., A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya (2017). “iFogSim: a toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments”. *Software: Practice and Experience* **47**:9, pp. 1275–1296.
- Gupta, R. A. and M.-Y. Chow (2008). *Networked Control Systems: Theory and Applications*. Springer-Verlag London.
- Hegazy, T. and M. Hefeeda (2015). “Industrial automation as a cloud service”. *IEEE Transactions on Parallel and Distributed Systems* **26**:10, pp. 2750–2763.
- Heilig, L., R. R. Negenborn, and S. Voß (2015). “Cloud-based intelligent transportation systems using model predictive control”. In: *International Conference on Computational Logistics*. Springer, pp. 464–477.
- Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2004). *Feedback control of computing systems*. Wiley Online Library.
- Herceg, M., M. Kvasnica, C. Jones, and M. Morari (2013). “Multi-Parametric Toolbox 3.0”. In: *Proc. of the European Control Conference*. Zürich, CH, pp. 502–510.

- Horn, C. and J. Krüger (2016). “Feasibility of connecting machinery and robots to industrial control services in the cloud”. In: *IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. Berlin, Germany, pp. 1–4.
- Hrovat, D., S. Di Cairano, H. E. Tseng, and I. V. Kolmanovsky (2012). “The development of model predictive control in automotive industry: a survey”. In: *IEEE International Conference on Control Applications*. Dubrovnik, Croatia, pp. 295–302.
- Hu, W., Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, and M. Satyanarayanan (2016). “Quantifying the impact of edge computing on mobile applications”. In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '16. ACM, Hong Kong, Hong Kong, 5:1–5:8. ISBN: 978-1-4503-4265-0.
- Inaltekin, H., M. Gorlatova, and M. Chiang (2018). “Virtualized control over fog: interplay between reliability and latency”. *IEEE Internet of Things Journal* **5**:6, pp. 5030–5045.
- Internet Systems Consortium (2019). *Internet domain survey*. Internet Systems Consortium. URL: <https://downloads.isc.org/www/survey/reports/2019/01/> (visited on 2021-02-20).
- Jasperneite, J., T. Sauter, and M. Wollschlaeger (2020). “Why we need automation models: handling complexity in industry 4.0 and the internet of things”. *IEEE Industrial Electronics Magazine* **14**:1, pp. 29–40.
- Kaneko, Y. and T. Ito (2016). “A reliable cloud-based feedback control system”. In: *IEEE 9th International Conference on Cloud Computing (CLOUD)*. San Francisco, CA, USA.
- Kang, J.-M., H. Bannazadeh, and A. Leon-Garcia (2013). “SAVI testbed: control and management of converged virtual ICT resources”. In: *IFIP/IEEE International Symposium on Integrated Network Management*. Ghent, Belgium, pp. 664–667.
- Khan, W. A., L. Wisniewski, D. Lang, and J. Jasperneite (2017). “Analysis of the requirements for offering industrie 4.0 applications as a cloud service”. In: *IEEE 26th international symposium on industrial electronics (ISIE)*. Edinburgh, UK, pp. 1181–1188.
- Kihl, M., A. Robertsson, M. Andersson, and B. Wittenmark (2008). “Control-theoretic analysis of admission control mechanisms for web server systems”. *World Wide Web* **11**:1, pp. 93–116.
- Kim, W.-j., K. Ji, and A. Srivastava (2006). “Network-based control with real-time prediction of delayed/lost sensor data”. *IEEE Transactions on Control Systems Technology* **14**:1, pp. 182–185.

- Leitner, P. and J. Cito (2016). “Patterns in the chaos—a study of performance variation and predictability in public IaaS clouds”. *ACM Transactions on Internet Technology (TOIT)*.
- Levine, W. S. and S. V. Raković, (Eds.) (2018). *Handbook of Model Predictive Control*. Birkhäuser Basel.
- Li, Q., Q. Hao, L. Xiao, and Z. Li (2009). “Adaptive management of virtualized resources in cloud computing using feedback control”. In: *First International Conference on Information Science and Engineering (ICISE)*. IEEE, Nanjing, China.
- Limón, D., I. Alvarado, T. Alamo, and E. F. Camacho (2010). “Robust tube-based MPC for tracking of constrained linear systems with additive disturbances”. *Journal of Process Control* **20**:3.
- Lu, Q., Y. Sun, Q. Zhou, and Z. Feng (2014). “New results on robust model predictive control for time-delay systems with input constraints”. *Journal of Applied Mathematics* **2014**.
- Luck, R. and A. Ray (1994). “Experimental verification of a delay compensation algorithm for integrated communication and control systems”. *International Journal of Control* **59**:6, pp. 1357–1372.
- Lyu, M., F. Biennier, and P. Ghodous (2019). “Control as a service architecture to support cloud-based and event-driven control application development”. In: *IEEE International Congress on Internet of Things (ICIOT)*. Milan, Italy, pp. 41–49.
- Ma, Y., C. Lu, B. Sinopoli, and S. Zeng (2020). “Exploring edge computing for multitier industrial control”. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **39**:11, pp. 3506–3518.
- Maciejowski, J. M. (2002). *Predictive control: with constraints*. Pearson Education.
- Magni, L., R. Scattolini, and M. Tanelli (2008). “Switched model predictive control for performance enhancement”. *International Journal of Control* **81**:12, pp. 1859–1869.
- Mahmoud, M. S. and Y. Xia (2020). *Cloud Control Systems: Analysis, Design and Estimation*. Academic Press.
- Mahmud, N., K. Sandström, and A. Vulgarakis (2014). “Evaluating industrial applicability of virtualization on a distributed multicore platform”. In: *Emerging Technology and Factory Automation (ETFA)*. IEEE, pp. 1–8.
- Malkowsky, S., J. Vieira, L. Liu, P. Harris, K. Nieman, N. Kundargi, I. C. Wong, F. Tufvesson, V. Öwall, and O. Edfors (2017). “The world’s first real-time testbed for Massive MIMO: design, implementation, and validation”. *IEEE Access* **5**, pp. 9073–9088.

- MATLAB (2020). *version 9.7.0 Matlab R2020b*. Ed. by I. The Mathworks. The MathWorks Inc., Natick, MA, USA.
- Mattingley, J. and S. Boyd (2012). “CVXGEN: a code generator for embedded convex optimization”. *Optimization and Engineering* **13**:1, pp. 1–27.
- Mattingley, J. and S. Boyd (2010). “Real-time convex optimization in signal processing”. *IEEE Signal Processing Magazine* **27**:3, pp. 50–61.
- Maxim, A., D. Copot, C. Copot, and C. M. Ionescu (2019). “The 5w’s for control as part of Industry 4.0: why, what, where, who, and when—a PID and MPC control perspective”. *Inventions* **4**:1. ISSN: 2411-5134.
- McCarthy, K. (2001). *World’s first 3G network live today*. URL: https://www.theregister.com/2001/10/01/worlds_first_3g_network_live/ (visited on 2021-02-20).
- Mell, P. and T. Grance (2011). *The NIST definition of cloud computing*. Special Publication (NIST SP), National Institute of Standards and Technology. Gaithersburg, MD, USA.
- Michalska, H. and D. Q. Mayne (1993). “Robust receding horizon control of constrained nonlinear systems”. *IEEE Transactions on Automatic Control* **38**:11, pp. 1623–1633.
- Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller (2013). “Playing Atari with deep reinforcement learning”. *Computing Research Repository (CoRR)* **abs/1312.5602**. arXiv: [1312.5602](http://arxiv.org/abs/1312.5602). URL: <http://arxiv.org/abs/1312.5602>.
- Mohanty, S. K., G. Premsankar, and M. di Francesco (2018). “An evaluation of open source serverless computing frameworks”. In: *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Nicosia, Cyprus, pp. 115–120.
- Mubeen, S., P. Nikolaidis, A. Didic, H. Pei-Breivold, K. Sandström, and M. Behnam (2017). “Delay mitigation in offloaded cloud controllers in industrial IoT”. *IEEE Access* **5**, pp. 4418–4430. ISSN: 2169-3536.
- Murray, D. G., F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi (2013). “Naiad: a timely dataflow system”. In: *24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, USA, pp. 439–455.
- Naik, N. (2017). “Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP”. In: *IEEE International Systems Engineering Symposium (ISSE)*. IEEE. Vienna, Austria, pp. 1–7.
- NI (2016). *University of Bristol and Lund University partner with NI to set world records in 5G wireless spectral efficiency using massive MIMO*. Ed. by NI. URL: <http://sine.ni.com/cs/app/doc/p/id/cs-17101#> (visited on 2021-06-02).

- Nilsson, J. (1998). *Real-Time Control Systems with Delays*. PhD thesis. Department of Automatic Control, Lund University.
- Nylander, T., J. Ruuskanen, K.-E. Årzén, and M. Maggio (2020). “Modeling of request cloning in cloud server systems using processor sharing”. In: *11th ACM/SPEC International Conference on Performance Engineering (ICPE)*. Edmonton, Canada, pp. 24–35.
- Nylander, T., M. Thelander Andrén, K.-E. Årzén, and M. Maggio (2018). “Cloud application predictability through integrated load-balancing and service time control”. In: *IEEE International Conference on Autonomic Computing (ICAC)*. Trento, Italy, pp. 51–60.
- Ohlsson, P. G., J. Svensson, and H. Blackman (2015). *Ericssons mobiltelefoner 1983-2001*. Roos & Tegnér. ISBN: 978-91-86691-98-1.
- Ong, C.-J., Z. Wang, and M. Dehghan (2015). “Characterization of switching sequences on system with dwell-time restriction for model predictive control”. In: *54th IEEE Conference on Decision and Control (CDC)*. Osaka, Japan, pp. 3657–3662.
- OpenFog Consortium (2017). *OpenFog reference architecture for fog computing*. Industry IoT Consortium. URL: https://iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf (visited on 2021-10-07).
- OpenJS Foundation and Node-RED contributors (2021). *Node-RED*. URL: <https://nodered.org> (visited on 2021-10-07).
- Park, H.-W., P. Wensing, and S. Kim (2015). “Online planning for autonomous running jumps over obstacles in high-speed quadrupeds”. In: *Proceedings of Robotics: Science and Systems*. Rome, Italy.
- Pelle, I., J. Czentye, J. Dóka, and B. Sonko (2019). “Towards latency sensitive cloud native applications: a performance study on AWS”. In: *IEEE 12th International Conference on Cloud Computing (CLOUD)*. Milan, Italy, pp. 272–280.
- Persson, L. (2019). *Autonomous and Cooperative Landings Using Model Predictive Control*. PhD thesis. KTH Royal Institute of Technology.
- Persson, P. and O. Angelsmark (2015). “Calvin—merging cloud and IoT”. *Procedia Computer Science* **52**, pp. 210–217.
- Picasso, B., D. De Vito, R. Scattolini, and P. Colaneri (2010). “An MPC approach to the design of two-layer hierarchical control systems”. *Automatica* **46**:5.
- Ploplys, N. J., P. A. Kawka, and A. G. Alleyne (2004). “Closed-loop control over wireless networks”. *IEEE Control Systems Magazine* **24**:3, pp. 58–71.

- Posthumus-Cloosterman, M. (2008). *Control over communication networks: Modeling, analysis, and synthesis*. PhD thesis. Ph. D. dissertation, Tech. Univ. Eindhoven, Eindhoven, The Netherlands.
- Qin, S. and T. A. Badgwell (2003). “A survey of industrial model predictive control technology”. *Control Engineering Practice* **11**:7, pp. 733–764. ISSN: 0967-0661.
- Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, et al. (2009). “ROS: an open-source robot operating system”. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan, p. 5.
- Rawlings, J. and D. Mayne (2009a). *Model Predictive Control: Theory and Design*. Nob Hill Pub. ISBN: 9780975937709. URL: https://books.google.se/books?id=3%5C_rfQQAACAAJ.
- Rawlings, J. B. and D. Q. Mayne (2009b). *Model Predictive Control: Theory and Design*. Nob Hill Pub. ISBN: 9780975937709. URL: https://books.google.se/books?id=3%5C_rfQQAACAAJ.
- Recht, B. (2018). “A tour of reinforcement learning: the view from continuous control”. *Computing Research Repository (CoRR)* **abs/1806.09460**. arXiv: 1806.09460. URL: <http://arxiv.org/abs/1806.09460>.
- Rojko, A. (2017). “Industry 4.0 concept: background and overview”. *International Journal of Interactive Mobile Technologies (iJIM)* **11**:5, pp. 77–90.
- Roosbeh, A., J. Soares, G. Q. Maguire, F. Wuhib, C. Padala, M. Mahloo, D. Turull, V. Yadhav, and D. Kostić (2018). “Software-defined “hardware” infrastructures: a survey on enabling technologies and open research directions”. *IEEE Communications Surveys & Tutorials* **20**:3, pp. 2454–2485.
- Rostedt, S. and D. V. Hart (2007). “Internals of the RT patch”. In: *Proceedings of the Linux symposium*. Vol. 2. Citeseer. Ottawa, Ontario Canada, pp. 161–172.
- Rüth, J., R. Glebke, K. Wehrle, V. Causevic, and S. Hirche (2018). “Towards in-network industrial feedback control”. In: *ACM Morning Workshop on In-Network Computing*. Budapest, Hungary. ISBN: 9781450359085.
- Ryden, M., K. Oh, A. Chandra, and J. Weissman (2014). “Nebula: distributed edge cloud for data intensive computing”. In: *IEEE International Conference on Cloud Engineering (IC2E)*, pp. 57–66.
- Saikrishna, P. and R. Pasumarthi (2016). “Multi-objective switching controller for cloud computing systems”. *Control Engineering Practice* **57**, pp. 72–83. ISSN: 0967-0661.

- Sapuntzakis, C. P., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum (2002). “Optimizing the migration of virtual computers”. *ACM SIGOPS Operating Systems Review* **36**:SI, pp. 377–390.
- Scattolini, R. and P. Colaneri (2007). “Hierarchical model predictive control”. In: *46th IEEE Conference on Decision and Control (CDC)*. DOI: [10.1109/CDC.2007.4434079](https://doi.org/10.1109/CDC.2007.4434079).
- Scattolini, R. (2009). “Architectures for distributed and hierarchical model predictive control – a review”. *Journal of Process Control* **19**:5, pp. 723–731.
- Schulz, P., M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel, A. Puschmann, A. Mitschele-Thiel, M. Muller, T. Elste, and M. Windisch (2017). “Latency critical IoT applications in 5G: perspective on the design of radio interface and network architecture”. *IEEE Communications Magazine* **55**:2, pp. 70–78. ISSN: 0163-6804.
- Science and Media Museum (2020). *A short history of the internet*. Science and Media Museum. URL: <https://www.scienceandmediamuseum.org.uk/objects-and-stories/short-history-internet> (visited on 2021-02-12).
- Sefidcon, A., W. John, M. Opsenica, and B. Skubic (2021). “The network compute fabric”. *Ericsson Technology Review* **7**. ISSN: 0014-0171.
- Seto, D., B. Krogh, L. Sha, and A. Chutinan (1998). “The simplex architecture for safe online control system upgrades”. In: *Proceedings of the 1998 American Control Conference (ACC)*. Vol. 6. Philadelphia, PA, USA. DOI: [10.1109/ACC.1998.703255](https://doi.org/10.1109/ACC.1998.703255).
- Sha, L. (2001). “Using simplicity to control complexity”. *IEEE Software* **4**, pp. 20–28.
- Shafi, M., A. F. Molisch, P. J. Smith, T. Haustein, P. Zhu, P. De Silva, F. Tufvesson, A. Benjebbour, and G. Wunder (2017). “5G: a tutorial overview of standards, trials, challenges, deployment, and practice”. *IEEE Journal on Selected Areas in Communications* **35**:6, pp. 1201–1221.
- Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis (2018). “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. *Science* **362**:6419, pp. 1140–1144. ISSN: 0036-8075.
- Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis (2017). “Mastering the game of Go without human knowledge”. *Nature* **550**, pp. 354–

- Simson L., G. and A. Harold (1999). *Architects of the Information Society : Thirty-Five Years of the Laboratory for Computer Science at MIT*. The MIT Press. ISBN: 9780262071963.
- Skarin, P. (2018). *Pyqpgen*. URL: <https://github.com/pskarin/pyqpgen> (visited on 2021-07-12).
- Skarin, P. and K.-E. Årzén (2021). “Explicit MPC recovery for cloud control systems”. In: *60th IEEE Conference on Decision and Control (CDC)*. Austin, TX, USA.
- Skarin, P., J. Eker, and K.-E. Årzén (2020a). “A cloud-enabled rate-switching MPC architecture”. In: *59th IEEE Conference on Decision and Control (CDC)*. Jeju, Korea (South).
- Skarin, P., J. Eker, and K.-E. Årzén (2020b). “Cloud-based model predictive control with variable horizon”. In: *21st International Federation of Automatic Control (IFAC) World Congress*. Berlin, Germany.
- Skarin, P., J. Eker, M. Kihl, and K.-E. Årzén (2019). “Cloud-assisted model predictive control”. In: *IEEE International Conference on Edge Computing (EDGE)*. Milano, Italy, pp. 110–112.
- Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2018). “Towards mission-critical control at the edge and over 5G”. In: *IEEE International Conference on Edge Computing (EDGE)*. San Francisco, USA, pp. 50–57. Best Paper Award.
- Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2020c). “Control-over-the-cloud: a performance study for cloud-native, critical control systems”. In: *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. Leicester, UK, pp. 57–66.
- Skarin, P., W. Tärneberg, K.-E. Årzén, and M. Kihl (2021). “Factory automation meets the cloud: Realizing resilient cloud-based optimal control for Industry 4.0”. *Submitted to IEEE Transactions on Industrial Informatics*.
- Škulj, G., R. Vrabčič, P. Butala, and A. Sluga (2013). “Statistical process control as a service: an industrial case study”. *Procedia CIRP* **7**, pp. 401–406.
- Stewart, B. T., A. N. Venkat, J. B. Rawlings, S. J. Wright, and G. Pannocchia (2010). “Cooperative distributed model predictive control”. *Systems & Control Letters* **59**:8, pp. 460–469. ISSN: 0167-6911.
- Sutton, R. S. and A. G. Barto (1998). *Introduction to Reinforcement Learning*. 1st. MIT Press, Cambridge, MA, USA. ISBN: 0262193981.
- Szydło, T., R. Brzoza-Wońc, J. Senderek, M. Windak, and C. Gniady (2017). “Flow-based programming for IoT leveraging fog computing”. In: *IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. Poznan, Poland, pp. 74–79.

- Tärneberg, W. (2019). *The confluence of cloud computing, 5G, and IoT in the fog*. PhD thesis. Lund University.
- Tärneberg, W., V. Chandrasekaran, and M. Humphrey (2016). “Experiences creating a framework for smart traffic control using AWS IoT”. In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. UCC '16. ACM, Shanghai, China, pp. 63–69. ISBN: 978-1-4503-4616-0.
- Tärneberg, W., M. Karaca, A. Robertsson, F. Tufvesson, and M. Kihl (2017a). “Utilizing massive mimo for the tactile internet: advantages and trade-offs”. In: *SECON Workshops - Robotic Wireless Networks*. IEEE. San Diego, CA, USA.
- Tärneberg, W., A. V. Papadopoulos, A. Mehta, J. Tordsson, and M. Kihl (2017b). “Distributed approach to the holistic resource management of a mobile cloud network”. In: *1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE. Madrid, Spain, pp. 51–60.
- Tipsuwan, Y. and M.-Y. Chow (2003). “Control methodologies in networked control systems”. *Control Engineering Practice* 11:10. Special Section on Control Methods for Telecommunication, pp. 1099–1111. ISSN: 0967-0661.
- Tobuschat, S., R. Ernst, A. Hamann, and D. Ziegenbein (2016). “System-level timing feasibility test for cyber-physical automotive systems”. In: *11th IEEE Symposium on Industrial Embedded Systems (SIES)*. Krakow, Poland, pp. 1–10.
- Vick, A., V. Vonásek, R. Pěnička, and J. Krüger (2015). “Robot control as a service — towards cloud-based motion planning and control for industrial robots”. In: *10th International Workshop on Robot Motion and Control (RoMoCo)*. IEEE. Poznan, Poland. DOI: [10.1109/RoMoCo.2015.7219710](https://doi.org/10.1109/RoMoCo.2015.7219710).
- Vick, A., J. Guhl, and J. Krüger (2016). “Model predictive control as a service—concept and architecture for use in cloud-based robot control”. In: *IEEE International Conference on Methods and Models in Automation and Robotics (MMAR)*. Miedzydroje, Poland.
- Villari, M., M. Fazio, S. Dustdar, O. Rana, and R. Ranjan (2016). “Osmotic computing: a new paradigm for edge/cloud integration”. *IEEE Cloud Computing* 3:6. ISSN: 2325-6095.
- Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, Í. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors (2020). “SciPy

- 1.0: Fundamental Algorithms for Scientific Computing in Python”. *Nature Methods* **17**, pp. 261–272.
- Vyatkin, V. (2011). “IEC 61499 as enabler of distributed and intelligent automation: state-of-the-art review”. *IEEE Transactions on Industrial Informatics* **7**:4, pp. 768–781.
- Wan, J., S. Tang, Z. Shu, D. Li, S. Wang, M. Imran, and A. V. Vasilakos (2016). “Software-defined industrial internet of things in the context of industry 4.0”. *IEEE Sensors Journal* **16**:20, pp. 7373–7380. ISSN: 1530-437X.
- Weil, S. A., S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn (2006). “Ceph: a scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, Seattle, WA, USA, pp. 307–320.
- Wildhagen, S. and F. Allgöwer (2020). “Rollout scheduling and control for distributed systems via tube MPC”. In: *IEEE Conference on Decision and Control*. Jeju, Korea (South).
- Xia, F. and W. Zhao (2007). “Flexible time-triggered sampling in smart sensor-based wireless control systems”. *Sensors* **7**:11, pp. 2548–2564.
- Xia, Y. (2012). “From networked control systems to cloud control systems”. In: *Proceedings of the 31st Chinese Control Conference*. Hefei, China.
- Xia, Y. (2015). “Cloud control systems”. *IEEE/CAA Journal of Automatica Sinica* **2**:2, pp. 134–142.
- Xu, Y. (2017). *LQG-Based Real-Time Scheduling and Control Codesign*. PhD thesis. Department of Automatic Control, Lund University.
- Yang, L. and S.-H. Yang (2007). “Multirate control in internet-based control systems”. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **37**:2.
- Yang, S. H., X. Chen, L. S. Tan, and L. Yang (2005). “Time delay and data loss compensation for internet-based process control systems”. *Transactions of the Institute of Measurement and Control* **27**:2, pp. 103–118.
- Yfoulis, C. A. and A. Gounaris (2009). “Honoring SLAs on cloud computing services: a control perspective”. In: *European Control Conference (ECC)*. EUCA.
- 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. ISSN: 1383-7621.
- Zakon, R. H. (2018). *Hobbes’ internet timeline*. Zakon Group LLC and OpenConf. URL: <https://www.zakon.org/robert/internet/timeline/> (visited on 2021-02-20).

- Zeilinger, M. N., C. N. Jones, and M. Morari (2010). “Robust stability properties of soft constrained MPC”. In: *49th IEEE Conference on Decision and Control (CDC)*. Atlanta, GA, USA, pp. 5276–5282.
- Zhang, L., S. Zhuang, and R. D. Braatz (2016). “Switched model predictive control of switched linear systems: feasibility, stability and robustness”. *Automatica* **67**thesis, pp. 8–21. ISSN: 0005-1098.
- Zhao, Y.-B., X.-M. Sun, J. Zhang, and P. Shi (2015). “Networked control systems: the communication basics and control methodologies”. *Mathematical Problems in Engineering* vol. **2015** (Article ID 639793), 9 pages. ISSN: 1024-123X.
- Zheng, Y., S. E. Li, K. Li, F. Borrelli, and J. K. Hedrick (2016). “Distributed model predictive control for heterogeneous vehicle platoons under unidirectional topologies”. *IEEE Transactions on Control Systems Technology* **25**:3, pp. 899–910.

A

Cost tables

These tables show the states and accumulated CLRE for the simulations summed up in Table 13.4. States and distributions are show in Figure 13.3, Figure 13.4, Table 13.2.

Method	States (processing,flight) and end time						
	s_{1,s_3} 16s	s_{1,s_4} 33s	s_{2,s_4} 35s	s_{1,s_4} 37s	s_{1,s_3} 56s	s_{2,s_3} 61s	s_{1,s_3} 88s
Adapt	0.00	1.64	1.73	1.74	2.42	2.45	3.13
Adapt [†]	0.00	2.20	2.38	2.42	2.67	2.67	3.32
22.2 Hz NDOL	0.01	1.39	1.45	1.48	1.88	1.95	2.28
22.2 Hz NDOL [†]	0.01	1.39	1.45	1.48	2.34	2.57	2.90
16.6 Hz NDOL	0.02	1.04	1.13	1.37	2.43	2.48	3.41
16.6 Hz NDOL [†]	0.02	1.04	1.13	1.37	2.43	2.48	3.41

Method	States (processing,flight) and end time						
	s_{2,s_3} 101s	s_{1,s_3} 105s	s_{2,s_3} 107s	s_{1,s_3} 116s	s_{2,s_3} 117s	s_{1,s_3} 145s	s_{2,s_3} 154s
Adapt	3.19	3.19	3.23	3.54	3.54	9.09	9.16
Adapt [†]	3.39	3.49	3.65	3.80	3.81	12.7	12.7
22.2 Hz NDOL	2.50	2.52	2.64	2.74	2.75	12.0	12.1
22.2 Hz NDOL [†]	3.12	3.14	3.26	3.36	3.37	209	209
16.6 Hz NDOL	3.88	3.91	4.01	4.42	4.46	14.4	14.6
16.6 Hz NDOL [†]	3.88	3.91	4.01	4.42	4.46	14.3	14.4

Method	States (processing,flight) and end time						
	s_1,s_3 163s	s_2,s_3 169s	s_1,s_3 170s	s_1,s_4 174s	s_2,s_4 175s	s_1,s_4 180s	s_2,s_4 202s
Adapt	9.24	9.31	9.31	9.99	9.99	10.2	10.9
Adapt [†]	13.0	13.0	13.0	13.2	13.2	13.3	14.1
22.2 Hz NDOL	12.2	12.3	12.3	12.7	12.7	13.1	13.6
22.2 Hz NDOL [†]	210	210	210	210	210	210	211
16.6 Hz NDOL	14.9	15.1	15.1	15.4	15.4	15.5	16.7
16.6 Hz NDOL [†]	14.8	15.0	15.0	15.3	15.3	15.4	16.5

Method	States (processing,flight) and end time						
	s_1,s_4 221s	s_2,s_4 221s	s_2,s_5 229s	s_1,s_5 232s	s_1,s_3 232s	s_2,s_3 283s	s_1,s_3 283s
Adapt	13.6	13.6	14.1	14.1	14.1	14.4	14.4
Adapt [†]	16.6	16.6	17.2	17.2	17.2	17.3	17.3
22.2 Hz NDOL	17.6	17.6	17.8	17.9	17.9	18.6	18.6
22.2 Hz NDOL [†]	304	304	304	305	305	305	305
16.6 Hz NDOL	19.9	19.9	20.1	20.4	20.4	22.3	22.3
16.6 Hz NDOL [†]	21.8	21.8	22.0	22.3	22.3	24.2	24.2

Method	States (processing,flight) and end time						
	s_2,s_3 292s	s_1,s_3 300s	s_1,s_4 303s	s_1,s_3 310s	s_2,s_3 318s	s_1,s_3 334s	s_2,s_3 389s
Adapt	14.4	14.6	14.6	14.6	14.7	15.7	15.9
Adapt [†]	17.4	17.6	17.9	18.2	18.2	19.0	19.1
22.2 Hz NDOL	18.8	18.9	18.9	19.1	19.2	20.1	20.8
22.2 Hz NDOL [†]	305	306	306	306	306	307	307
16.6 Hz NDOL	22.8	23.1	23.2	23.5	23.8	25.1	27.0
16.6 Hz NDOL [†]	24.7	25.0	25.1	25.4	25.7	27.0	28.8

Appendix A. Cost tables

Method	States (processing,flight) and end time	
	s_1, s_3 389s	s_2, s_3 401s
Adapt	15.9	15.9
Adapt [†]	19.1	19.2
22.2 Hz NDOL	20.8	20.9
22.2 Hz NDOL [†]	307	308
16.6 Hz NDOL	27.0	27.7
16.6 Hz NDOL [†]	28.8	29.6