Disturbance Rejection and Control in Web Servers

Kjaer, Martin Ansbjerg

2009

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

Disturbance Rejection and Control in Web Servers

Disturbance Rejection and Control in Web Servers

Martin Ansbjerg Kjær

Department of Automatic Control
Lund University
Lund, 2009

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--1086--SE

Printed in Sweden by MediaTryck.
Lund 2009

Abstract

An important factor for a user of web sites on the Internet is the duration of time between the request of a web page until an answer has been returned. If this response time is too long, the user is likely to abandon the web site and search for other providers of the service. To avoid this loss of users, it is important for the web site operator to assure that users are treated sufficiently fast. On the other hand, it is also important to minimize the effort to optimize profit. As these objectives often are contradictory, an acceptable target response–time that can be formulated. The resources are allocated in a manner that ensures that long response times do not occur, while, at the same time, using as little resources as possible to not overprovision.

The work presented in this doctoral thesis takes a control–theoretic perspective to solve this problem. The resources are considered as the control input, and the response time as the main output. Several disturbances affect the system, such as the arrival rate of requests to the web site. A testbed was designed to allow repeatable experiments with different controller implementations. A server was instrumented with sensors and actuators to handle requests from 12 client computers with capability for changing work loads.

On the theoretical side, a model of a web server is presented in this thesis. It explicitly models a specific sensor implementation where buffering occurs in the computer prior to the sensor. As a result, the measurement of the arrival rate becomes state dependent under high load. This property turns out to have some undesirable effects on the controlled system. The model was capable of predicting the behavior of the testbed quite well.

Based on the presented model, analysis shows that feed–forward controllers suggested in the literature can lead to instability under certain circumstances at high load. This has not been reported earlier, but is in this doctoral thesis demonstrated by both simulations and experiments. The analysis explains why and when the instability arises.

In the attempt to predict future response–times this thesis also presents a feedback based prediction scheme. Comparisons between earlier predictions to the real response–times are used to correct a model based response time prediction. The prediction scheme is applied to a controller to compensate for disturbances before the effect propagates to the response time. The method improves the transient response in the case of sudden changes in the arrival rate of requests.

This doctoral thesis also presents work on a control solution for reserving CPU capacity for a given process or a given group of processes

on a computer system. The method uses only existing operating–system infrastructure, and achieves the desired CPU capacity in a soft real–time manner.

Preface

I was presented with the subject of queuing systems at the trail lecture for appointment as an associate professor of Anders Robertsson in June 2005. The presentation introduced the field and also related the subject to other fields such as high–way queuing modeling. What I found fascinating about the subject was that no consensus was really found in the modeling, and this is valid to some degree even today. According to the perspective one take, the problem can be considered as a continuous system, as a stochastic system, as a discrete–time system (with fixed time intervals), or as an event driven system. One can focus on averaged or expected values, or one can operate on exact variables. Any combination is of course also possible. Furthermore, the relations between variables are often nonlinear in a way that challenges the intuitive understanding, and it reveals even more challenges for control design.

Control of these systems has been around for many years, but not explicitly treated from a control perspective, since online adjustments are often designed in an *ad hoc* manner and without concern for formal proofs. All together, this field holds a large potential for further control research both within the theoretical field, but also from an application point of view by *e.g.* improving the transient responses.

I hope that the reader of this thesis shares my enthusiasm for the field, and will enjoy my contribution.

Martin Ansbjerg Kjær, September 2009.

Acknowledgments

First, I cannot thank my main supervisor, Anders Robertsson, enough. Anders is probably the most optimistic and positive person I know. Presented to the terrifying problem that the server of our lab starts to self–oscillate under certain circumstances, Anders joyfully responds: *What an interesting control–problem!*, something it indeed turned out to be. Also the capacity to reduce everything, from skiing to bicycling, to fundamentally being a control problem is a great source of amusement. I also thank Anders for the fine support and guidance throughout my whole time as a Ph.D. student.

Gratitude is paid to my co–supervisor within the Telecommunication field, Maria Kihl. Our numerous discussions have been an important accelerator in my journey towards the understanding of queuing systems. A few open questions still remain, which we have not been able to agree on: Is it politically correct, according to good business, to denote the clients (remember, clients=costumers=money) as *disturbances*? Is it acceptable to formulate an optimization criterion as *holding clients back in the system* in order to save CPU capacity? Also, I acknowledge Maria for the encouraging remark when I "discovered" that queues are nonlinear: "*Hey, we've known that for years!*".

Karl-Erik Årzén, my second co–supervisor, I acknowledge for his impressive ability to see things in a larger perspective and point out directions in the jungle of research. Karl–Erik amazes me with his clear sight of where to go—both regarding research and where to enjoy a quiet drink after a hard days work in a foreign town.

The honor for the fabulous computer–infrastructure at the department, designed and maintained by Leif Andersson and Anders Blomdell, cannot be overrated. Designing network experiments (with network and over the network) was simplified significantly by their work. The secretaries of the department Britt-Marie Mårtensson, Eva Schildt, Eva Westin, Agneta Tuszynski, and Ingrid Nilsson, are the fundamental glue keeping

the department together, regarding paper work, accounting, remembering important dates, ensuring that Ph.D. students do get married, and many other important aspects. The librarian Lisbeth R. Karlsson at IT-biblioteket Katrinebjerg is greatly acknowledged for her outstanding perseverance in locating papers of older date. It has been a privilege to work in the same building as the father of control in Sweden, Karl Johan Åström. Facing an unsolvable problem, I often consulted Karl Johan at the coffee break, and just as often his answer was in the line of "*Why don't you try ...— I used it to solve a similar problem 30 years ago*". The whole Department of Automatic Control, including former employees, I acknowledge for the great working environment, intellectually stimulating atmosphere, and friendly attitude.

Thanks to the room mates that I have had the opportunity to share office facilities with over the years: Johan Bengtsson and Toivo Perby Henningsson in Lund, and João Fernandes, Weishan Zhang, and Kristian E. Kjær in Århus.

During my study of communication systems and queuing theory I have had the pleasure of working together and having discussions with some interesting and friendly persons from the Department of Telecommunication at LTH. I especially thank Christian Nyberg and Mikael Andersson.

The Department of Computer Science (DAIMI) at the University of Aarhus is acknowledged for their open mind and hospitality. I appreciate that I was invited in, almost from the street, and given an Internet connection and a coffee mug (all an engineer needs) for more than a year. Also, I thank the staff of the department, both academic and administrative, for accepting me into their social activities, especially around the coffee table.

Thanks to the people that have spent hours proof reading this manuscript: My sister Maren Ansbjerg Kjær, Björn Wittenmark, Per-Ola Larsson, Leif Andersson, Eva Westin, and Mikael Lindberg. Other people by whom I have found great support, morally and technically, are Oskar Nilsson and Brad Schofield.

This work has been founded by the Swedish Research Council, under project 621-2006-5522 and the Lund Center for Control of Complex Engineering Systems (LCCC).

I greatly acknowledge my daughter Karoline for allowing me room for thinking. She really did a great job of *NOT SLEEPING* during the evenings bed–time ritual, giving my brain plenty of time for creative thinking.

Finally, I thank all of my family, and in particular my wife, for encouragement and support.

Martin

Contents

1

Introduction

1.1 Motivation

Automatic adaptation to changes in the behavior of the surroundings of a computer system has gained increasing interest from both academia and software–system suppliers to provide reliable service to costumers while maintaining a low operation cost. In the last years, the environmental effects of computing have also come into perspective, something which encourages to optimize the performance even further.

Computer systems are often operating in environments with many unknown interactions caused by the unpredictable behavior of clients, other computer systems, or even other computer programs. The offered, or requested, traffic to a network can change abnormally with very short notice. If large amounts of data have to pass through the network, the load on for example a news site can increase suddenly as a response to a big news event, or a cell–phone router can be overloaded temporarily just after midnight of New Year's Eve. Some changes are predictable, such as the daily variation of requests to a mail server, while others are totally unpredictable. To handle such large variations, hardware of computer systems are often over–dimensioned compared to normal load demands, resulting in a lot of wasted resources. To optimize the resource consumption, more advanced control algorithms are investigated, and here, the field of automatic control has a lot to offer. It is generally known that a possible hazard when tuning for too fast and accurate control is loss in robustness and in worst case instability, which has devastating effects on the performance. Therefore, modeling and analysis of the behavior of web–server systems have become more and more important.

This thesis mainly considers response–time control of web servers, which present challenging problems on both modeling, analysis, and control design.

1.2 Contribution of the Thesis

Modeling, Control Design, and Analysis

The Chapters 4, 5, and 6 study a problem where a web server is instrumented in an inexpedient manner, where only a limited part of the buffering proceeding the web server is included. This appears to have some dramatic effects for the control design, since what would be assumed to be a measurement of a disturbance turns out to be state dependent, and a zero enters the transfer function describing the relation from the control signal to the control output. A stability analysis is presented indicating that if the system is designed without taking the dynamics of the unmodeled buffers into account, the system can become unstable for certain choices of control parameters. A redesign of the control strategy to improve the robustness is suggested. The results from the analysis are verified by both simulations and experiments.

The subject is relevant, as the amount of queuing outside the application is not always known and often neglected. The results presented in these chapters show the danger by such a simplification.

Related Publications

Kjær, M. A. and A. Robertsson (2009): "Effects of neglecting buffers in feed–forward design for web servers." In *Proc. Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'09)*, pp. 61–68. San Francisco, CA.

Kjær, M. A. and A. Robertsson (2010): "Analysis of buffer delay in web–server control." In *Proc. American Control Conference (ACC'10)*. IEEE, Baltimore, Maryland. Submitted.

In both papers, Martin Ansbjerg Kjær conducted the modeling and stability analysis, developed the control structure as well as the experimental testbed, and conducted the tuning and the evaluation.

Improved Feed–Forward by Prediction

Several feed–forward strategies to improve the disturbance rejection in web servers have been suggested in the literature, but a majority of them have been based on queuing–theoretic expressions, which require averaging over a long time to be implemented. Furthermore, they rely on offline estimation of a certain parameter. The work presented in Chapters 4 and 5 indicates that the stability of the whole system depends on the choice of this specific parameter, if the arriving requests are queued before measurements are taken. The results in Chapter 7 presents an alternative

method to estimate the unknown parameter online using a feedback based prediction method. Earlier work presented in the field were based on offline estimations, which are not as robust to changes, and therefore result in less attractive transient performance.

Related Publications

Kjær, M. A., M. Kihl, and A. Robertsson (2007): "Response-time control of a single server queue." In *Proc. 46th IEEE Conference on Decision and Control (CDC'07)*, pp. 3812–3817. New Orleans, LA.

Kjær, M. A., M. Kihl, and A. Robertsson (2008): "Response-time control of a processor-sharing system using virtualized server environments." In *Proc. 17th IFAC World Congress, pp. 3612–3618*. Seoul, Korea.

Kjær, M. A., M. Kihl, and A. Robertsson (2009): "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers." *Network and Service Management, IEEE Trans. on*. Submitted.

In [Kjær *et al.*, 2007], Martin Ansbjerg Kjær developed the control structure, Maria Kihl developed the simulation program, and Martin Ansbjerg Kjær conducted the tuning and the evaluation.

In [Kjær *et al.*, 2008], Martin Ansbjerg Kjær developed the control structure, developed the simulation program, and conducted the tuning and the evaluation.

In [Kjær *et al.*, 2009], Martin Ansbjerg Kjær developed the control structure, developed the experimental testbed, and conducted the tuning and the evaluation.

Nice Resource–Reservation

The results described in Chapter 8 aim to obtain CPU capacity separation between different tasks in a computer system while keeping the overhead to a minimum. A feedback based method is used to achieve CPU capacity reservation on a kernel level, thus avoiding the need to make modifications to the applications. The implementation makes use of the Linux prioritizing scheme to assure a specified amount of CPU capacity to a given task. The CPU reservations are obtained using existing operating–system infrastructure.

Related Publications

Ohlin, M. and M. A. Kjær (2007): "Nice resource reservations in Linux." In *Proceedings, Second IEEE International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'07)*, pp. 20–26. Munich, Germany.

Martin Ohlin developed and implemented the CPUreservation algorithm and conducted the initial tests using infinite while loops. Martin Ansbjerg Kjær designed the web server testbed and conduced the web server experiments. All material on the subject of *Nice Resource Reservations in Linux* is presented in this thesis with the acceptance of Martin Ohlin.

Other Contributions

Other contributions by the author, not included in this thesis are:

Kjær, M. A. (2005): "Active stabilization of thermoacoustic oscillation." Licentiate Thesis ISRN LUTFD2/TFRT--3239--SE. Department of Automatic Control, Lund University, Sweden.

Kjær, M. A., R. Johansson, and A. Robertsson (2006): "Active control of thermoacoustic oscillation." In *Proceedings of the IEEE International Conference on Control Applications*, pp. 2480–2485. Munich, Germany.

2

Background

The scope of this chapter is to present relevant background material that assists the readers accustomed to control theory to gain a deeper understanding of computer systems and queuing systems. Therefore, this chapter will not present any control–theoretical issues, but a great deal on both queuing theory and computer–related issues. Some subjects are treated in detail, because they are used in later chapters, while other subjects are presented to supply a more general view.

The background to this thesis is based on many disciplines and areas. First, computer systems are set in the context of control theory. Different concepts of computer and telecommunication areas are presented, since these are relevant for later investigations. Specific methods for instrumentation are also discussed. Finally, a section is devoted to discuss how control is used in related areas of computer systems.

Computers are often divided into *hardware* and *software*. The hardware is the physical components in the computer, such as the Central Processing Unit (CPU), the memory, and the hard–disk. The software is the commands which are executed on the physical hardware. As the complexity of both the hardware and the software increases, the two layers can be hard to distinguish. Often, instructions are implemented in software to ensure that they can easily be redesigned at a later stage, but they could also be implemented in hardware to obtain faster execution. Furthermore, both the software and the hardware can be divided into

several categories. Several physical computers can be united to solve a common task. Examples are super computers, large web sites, and large databases. The software is often divided into operating systems and applications, and even into user applications. Computing across multiple physical computers requires that communication is taken into account both in the software and in the hardware. For instance, a web server can forward requests to an application server, which in turn calls a database server. The database server can be running on two physical computers. The term *computer system* is used as an abstraction covering the unified functionality of the hardware, of the software, and of the infrastructure. A computer system has one more important factor, the users. Computer systems may have few or many users, and these users can have very different behaviors, and their interaction with the computer system generally results in unpredictable requirements of the computer system. Only a few users occupy a super computer at a given time, but the complexity of computations can make the execution quite unpredictable. In a news site, a large amount of readers request articles at a fairly random pattern. The operators can have different variables to tune the performance. Some can be tuned automatically while others are to be tuned manually. Automatic control can supply tools to assist the operator in adjusting the variables.

2.1 Computer Systems and Control Theory

Even though dynamics are not considered in traditional computer engineering, computer systems are dynamical systems, like other systems traditionally handled by control, such as robots, airplanes, and chemical reactors. It takes time from when a certain variable has changed until the full reaction can be observed. Like other dynamical systems, computer systems also follow certain physical laws, for example conservation laws, but other dynamic properties are of more synthetic nature caused by design choices.

In control theory, a system is often described in terms of inputs, outputs (measurements), disturbances and states, as illustrated in Fig. 2.1. Computer systems are seldom expressed in these terms. The following will give a general introduction to what these terms can cover when describing computer systems.

Measurements

Often computer systems are not designed with any control implementation in mind, and therefore, measurements of relevant variables can be a problem. In some cases, the variables are not available for the controller,

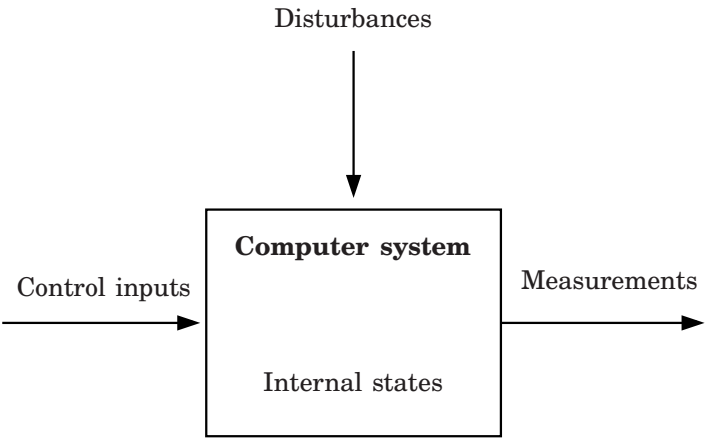

Figure 2.1 A computer system from a control perspective.

because the variables are hidden in lower software layers, for example, in the operating system. It may seem trivial to retrieve the time of an arrival of a mail to a mail server, but in fact, this is not trivial. One could argue that because the mail arrives via a TCP connection, the arrival time can be defined as the time when the connection is being requested. Here, the problem is, that the operating system is not necessarily aware of a connection request being associated with a mail. It may as well have been a HTTP request. Another definition of the arrival time could be the time where the full request has been delivered from the TCP layer to the mail program. If the network load is high, some parts of the mail could have spent much time in the buffer, before all parts of the mail have arrived, and this definition is thus also problematic. Furthermore, software systems are often designed in a layered structure, where layers are not supposed to know too much of what is going on inside the other layers (this layer structure is very beneficial from a system design point of view), so the relevant information to obtain a given measurement may not be available at all. Examples of layered architectures are the Open System Interconnection reference model (OSI) and the TCP/IP model, which are discussed in [Tanenbaum, 1996].

Very often, measurements of variables in computer systems can be inflicted with some delay not usually observed in other control systems. For instance, the measurement of the response times of a server has a variable delay, since the measurement of the response time of a particular job can only be measured when the job has been completed. Since this variable is treated as state dependent, it must be assumed that it can

change over time, which means that the measurement has a varying time delay. A similar problem exists in the congestion control of the Internet, where the response time in some cases is treated as a measure of the level of congestion. In this situation, the delay of the measurement will be state dependent, since the output of the control system actually is the delay. This type of state dependent delays is usually not covered by traditional control–analysis and control–design methods.

Control Inputs

Many variables can be tuned in a computer system. This is often handled manually by the system administrator, through text files or dialog boxes. Many are set to some default values by the software designer and never changed during operation. Some values require privileged rights to avoid unauthorized users to cause problems for the computer system. Other variables are not accessible when the system has been compiled and implemented. Examples of easily accessible inputs are the `nice`value of the UNIX system (the owner can indicate the relative priority of a process with respect to the other processes on the system), or the maximum number of jobs which are allowed to access a web server simultaneously. From a control point of view, it is extremely important whether these variables can be changed online or not. The `nice`value can be changed online as the process is running, but in the case of an Apache server, the maximal number of jobs allowed is defined when the server is initiated, and cannot be changed during operation. For this to be a control input, the source code of the server must be modified, as done in *e.g.*, [Lindegren, 2008]. The maximum length of buffers can also serve as control signals. These buffers can be hidden in the operating system, and are seldom accessible, in particular not during operation. Instrumenting these kinds of actuators are therefore not trivial, if at all feasible.

States

Many variables in a computer system can be dynamical; queue lengths, number of processes and threads, flow rates on networks and data buses, load indicators, and instantaneous throughput. Often, the dynamics of these variables are not considered during system design, and the variables are not directly measurable.

Disturbances

Disturbances in control terminology are those variables (inputs), which are independent of the states and which cannot be affected by the control signal, but which change in an unpredictable manner. Disturbances in computer systems can be caused by user interactions, such as starting

programs on a computer or requesting a service from a server. Disturbances can also be generated by the computer system itself, for example when backup procedures or virus scanners are started. Even though these tasks are not random or unpredictable by nature, these types of actions can be impossible to keep track of, and are therefore considered as disturbances.

Limitations

As most other systems, computer systems experience limitations. Buffers have limited sizes, negative numbers of processes in the system are not feasible, and priority levels are restricted to a finite interval. These limitations are often defined during system design, or in some cases, they can be defined by user (at least during initializion). The levels of these limitations are often fixed and known (or at least, it is feasible to obtain knowledge of their values). Other limitations are not known and may not even be constant. The amount of data that a network can carry before congesting is not a known value. The amount of active jobs a system can handle before paging becomes a problem, is also a limit which is so hard to predict that it must be considered as unknown. These kinds of limitations are difficult to handle. Often, these problems can be solved by some probing strategy, like for example in the design of the Internet congestion–control. See Section 2.7 for a more detailed discussion of the congestion control.

Models

Computers can be viewed from many different angles, resulting in different modeling approaches. The computer consists of physical devices like resistors and transistors which are continuous on a low–level scale. Neglecting the dynamics of the capacitors and transistors, the system becomes discrete. If a variable is represented by sufficiently many bits, the quantification can be neglected, and the variable can be viewed as real valued. On the scheduling level, only one task has access to the CPU at a given time, and the variable describing the tasks access to the CPU is thus a discrete, continuous–time variable, which is zero when the process occupies the CPU and one when some other process occupies the CPU. On a larger time scale, several tasks have been given access to the CPU, and a given task has then been given a fraction of the total amount of computation done. On this time scale, the CPU–fraction can be seen as a continuous discrete–time variable between 0% and 100%. Likewise, the amount of jobs in a buffer can be seen as a discrete continuous–time value, describing the absolute amount of jobs at any time. This variable can be rather fluctuating and its expected value may be of more interest. This value can be defined as the average over a time interval, resulting in a

real discrete–time variable, or it may be found by a flow–model approach, resulting in a real continuous–time variable. Furthermore, the dynamics can be neglected completely and a statistical approach can be taken, namely the field of queuing theory (to be described later). As these examples illustrate, the modeling of computer systems depends on what level of accuracy and time scale is under investigation. Often, this choice is restricted since modeling tools not always exist for a given accuracy level.

In other systems, such as mechanical systems and chemical systems, relations between variables can be expressed by known mathematical relations, such as Newton's laws and Ohm's law. Similar relations are hard to find for computer systems, and often more data–driven approaches are taken. System identification has been used in many cases to assist the development of local controllers and to determine stability and robustness properties. Examples are [Lu *et al.*, 2001; Hellerstein *et al.*, 2004; Lu *et al.*, 2006].

2.2 Virtualization and Resource Reservation

The history of virtualization and time–sharing systems dates back to the late 1950s and early 1960s, see *e.g.* [Strachey, 1959]. In time–sharing systems different applications run on the same hardware, and the time–share mechanism distributes the physical resources between the applications according to some scheduling method. The applications are aware that they share the resources with other applications, and can make use of specific instructions to optimize the use of the physical resources. An application running in a virtualized system senses that it is running on a physical system, even though this is not the case. The virtual system just offers the same functionality as a physical system. The physical system can run several such virtual machines, and share the physical resources between them. Here, even operating systems can be seen as an application, which can be run in an isolated virtual machine. This is not possible for normal time–sharing systems. A clear formulation of the differences between these two methods were already formulated in 1969 by Madnick:

> *A CTSS* [Editor's note: Conventional Time–Sharing System] *characteristically provides a software interface to the user, whereas a VMTSS* [Editor's note: Virtual Machine Time–Sharing System] *presents the user with a simulated hardware interface to a virtual computer*. [Madnick, 1969]

This is still a good description.

Methods of resource reservation can be seen as an underlying mechanism to time–sharing systems, where the resource reservation is the

method to allocate the physical resources.

The differences between time–sharing system, resource reservation systems, and virtualized systems are of more computer–science nature, and are not really relevant for the work presented in this thesis. They all give the same functionality; physical resources are assigned to an application (or a group of applications) by an underlying mechanism, which can be altered by a (privileged) user. Therefore, the terms are used interchangeably in many places. A solution denoted as a virtualized system may be implemented as a time–share system with resource–reservation techniques, but it may just as well be implemented by virtualization techniques. In real–life implementations, the actual choice of technique will be important in relation to other factors, such as scalability, robustness, portability, and others, but because this is not the scope of this thesis, the more casual approach to virtualization and time–sharing is taken.

Mile–stone results on virtualization were delivered by the M44/44X project, presenting solutions on memory virtualization vs. handcrafted manual memory–allocation, paging problems, time–share solutions, and other issues [O'Neill, 1967; Shils, 1968; Sayre, 1969]. Even though formal control–theoretic methods were not utilized explicitly, online measurements of performance indicators were used for feedback in many cases to adjust performance during execution. These control mechanisms were designed and tuned *ad hoc* by intuition and engineering experience, and generally improved the robustness and performance significantly. An example of this is described in more details later in this thesis (Section 2.7). Today, resource reservation and virtualization have become important tools for modern IT–systems. For example, an Internet host (*e.g.* a web hotel) guarantees to supply a certain amount of resources to a number of service providers (*e.g.* web shops). Often several service providers are hosted on the same hardware, but the host must guarantee that each service provider receives the agreed amount of resources, despite the behavior of the other service providers. The specific type of resources can be one or more of the following; network bandwidth, database access, memory allocation, CPU capacity, and many more. Another example is when a movie player on a PC needs a certain amount of CPU capacity while a virus scanner runs in the background.

On a single computer, the exact decision of how the resources are split between the applications is often left to the operating system. In most cases, this can be seen as an advantage because it is not normally known exactly how important they are in relation to each other. For example, it is not trivial to determine how important the mail server is compared to the web server. However, in some cases, it would be advantageous if there existed a mechanism to specify exactly how important different tasks (or groups of tasks) are compared to each other.

When the a surplus of resources exists, the different applications running on the same hardware are not restricted in their operation, and they do not sense the other applications. If the resources are limited, or have a cost, the application will have to share the available resources, and now the behavior of one application will have affect on the other applications. Some applications are sensitive to variations or limitations in the resources given to them, and in such cases virtualization is a powerful tool.

Reservation of CPU Capacity

The concept of reservation–based scheduling has been called *fair–share scheduling* [Essick, 1990; Kay and Lauder, 1988; Henry, 1984] but is also known under the name *proportional–share scheduling* [Fong and Squillante, 1995; Stoica and Abdel-Wahab, 1995; Waldspurger and Weihl, 1995a; Waldspurger and Weihl, 1995b]. A good summary of this field with more details can be found in [de Jongh, 2002].

An early attempt to use the operating–system architecture for reservation–based scheduling is the *Watson Share scheduler* [Moruzzi and Rose, 1991]. It was implemented on top of a standard AIX operating system at the Compute Power Server Cluster at IBM, where the `nice` value was changed to enforce CPU capacity. It is also mentioned in [Hellerstein, 2004] and [Hellerstein *et al.*, 2005] as something that can be done in theory in UNIX, but is complicated in practice because of the non–linear relationship between `nice`, the number of processes, and the CPU capacity received. Provided that the number of jobs in the system is fixed, and that they are all present from the same time onward, a deterministic analysis of the steady state shares is possible. [Hellerstein, 1993] shows how this can be used to statically calculate the base priorities on a uni–processor in the presence of *decay–usage scheduling* in UNIX. [Epema, 1998] extends this analysis to the multiprocessor case.

Xen

Xen is a virtual–computer system, where several virtual computers, containers in Xen terminology, share the physical resources of a computer [Xen, 2009]. On top of the physical hardware is the Xen scheduler, which divides resources between the containers. Each container resembles a computer system, where individual operating systems and applications can run independently of the other containers. This implies that that all resources are virtualized. The basic container, container 0, has special management privileges, and is created when the system boots. From this domain, new domains can be created and managed. Xen is an open–source project which has proved suitable for online adjustment of resources in different applications, as for example [Xu *et al.*, 2006; Wang *et al.*, 2007].

VMware

VMware is a commercial virtualization–system which creates virtual computers on top of existing operating systems, see [VMware, 2009].

CKRM

Class–based Kernel Resource Management (CKRM) aims at providing differentiated services to resources in Linux such as CPU capacity, memory pages, I/O, and incoming network bandwidth [CKRM, 2009]. Parts of this project is used in "SuSE Linux Enterprise Server 9", not the CPU controller though.

Control Groups

Control Groups is a project under Linux to facilitate grouping of processes in the Linux kernel, supported from kernel 2.6.24 and onward. The aim is to make the grouping as easy and logical as possible, requiring a minimum of kernel–specific knowledge. Groups ordered with the Control–Groups functionality will be denoted CGroups. CGroups are ordered in a tree–like fashion where CGroups are sub–groups of other CGroups. This structure is implemented by mounting a Control–Groups file–system, and building a directory tree that matches that of the desired CGroup tree, as illustrated in Fig. 2.2. To associate a process to a CGroup, the process ID is written to the `tasks`–file of the relevant CGroup. This file structure is actually virtual; all files and directories exist only in the memory, but this is of no importance from a user perspective. See [Linux Headquarters, 2008b] for more details.

Different subsystems can be added to the Control–Groups functionality:

cpu alters the way the scheduler priorities the CPU capacity. The scheduler assigns CPU capacity according to a fair–scheduling principle; it tries to divide resources equally between the CGroups (note that the scheduler now distributes resources between CGroups and not between processes, as would have been the case without the Control–Groups functionality). By using the CPU–subsystem, the user can define ratios between the CGroups to alter how the CPU capacity are distributed. This is done by writing *share*–values into CGroup–specific files (named `cpu.share`). The processes of a CGroup are scheduled according to a fair–scheduler policy, which cannot be affected by the user, see [Linux Headquarters, 2008a].

cpuacct accounts for the time that the CPU has been allocated, and an accounting value is updated each time any process in the CGroup has accessed the CPU. The accounting value is available in the CGroup–file `cpuacct.usage`.

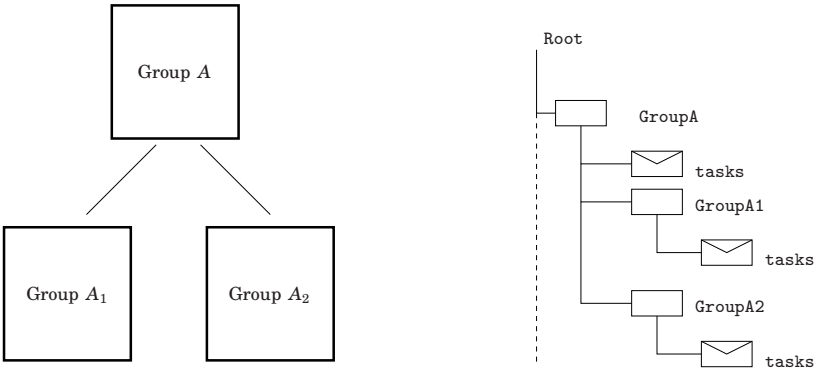

Figure 2.2 Illustration of a CGroup tree (left) and the corresponding Control–Groups implementation as a file–structure (right). CGroups A_1 and A_2 are independent CGroups, both a sub–CGroup to CGroup A. This corresponds to the Control–Groups implementation where the directories `GroupA1` and `GroupA2` are independent, but both are a sub–directories of `GroupA`. Process–IDs are written into the `tasks`–files to associate the process to the CGroup.

cpuset is used to schedule groups of physical CPUs. See [Linux Headquarters, 2008b] for more details.

memory allows the memory to be shared between CGroups. See [Linux Headquarters, 2009].

Even if the Control Groups system holds tools for virtualization of several types of resources, it cannot yet perform full virtualization.

2.3 Web Servers

The first web server and corresponding browser (to the authors knowledge) were developed at CERN around 1990 to assist scientists in sharing information [World Wide Web Consortium, 2009a; World Wide Web Consortium, 2009b]. Today, web servers are widely used on the Internet providing different services to a range of clients throughout the world. Examples are Internet shops, news sites, Internet banking, but also smaller personal sites used by private persons to share pictures, blogs, files, and much more.

A client requests a certain file, denoted *Uniform Resource Locator* (URL), from the web server, and the server responds with the requested file or some error message if it was not able to deliver the requested file. The protocol for communication is *HyperText Transfer Protocol* (HTTP),

which encapsulates messages and data in packets with additional information, related to the sender, the language, and the data flow.

The requested information can create bottlenecks at different places in the physical computer hosting the web server, depending on the type of information requested. If the web server often serves downloads of large–sized files, the I/O operations may be overloaded and create a bottleneck. This can in some cases be solved by cashing data, but this requires physical memory, which then can become a bottleneck. Other types of requests require heavy computational work from the CPU on the computer hosting the web server. This is often the case with dynamically generated web pages, such as PHP and CGI scripts. Here the CPU can become the bottleneck. If the access network of the server is slow, it can congest and create a bottleneck. The web server is capable of handling the requests, but not able to return them sufficiently fast to the client, because the network in between cannot handle the load. The main point is that different kinds of traffic can cause bottleneck and thereby poor performance. The bottlenecks can appear in many places in the computer system, and it can be difficult to predict where to focus the investment on physical resources, since the bottleneck depends so highly on the information to be requested.

The most common web server today is the Apache server which hosts around 52% of the sites world wide [Netcraft, 2009]. Apache is an open–source server, distributed and maintained by a community of developers under the *Apache Software Foundation*. It has a modular structure which allows the user to include or exclude different functionalities. A main focus of the Apache server is the operational stability. The *Internet Information Server* (IIS) from Microsoft is the second largest on the Internet, hosting around 33% of the sites [Netcraft, 2009]. This server is not an open–source project. Also Google is active on the web–server market, but with a remarkably lower market share of roughly 5% [Netcraft, 2009].

The Apache Server

Apache can run on various platforms, such as Linux, Windows, and several UNIX and BSD systems. Operating systems handle threads and processes in different ways. Therefore, operating–system specific modules are used as the interface between the operating system and the Apache server, while the rest of the Apache remains (almost) independent of the operating system. The `mpm_winnt` is the default choice for Windows applications, where requests are handled by threads contained in a single process. The `PreFork` module is often used for Linux and UNIX applications. Here, each process contains only one thread, and thus, a process handles only one request at a time. The name *pre–fork* indicates that Apache forks processes before they are needed, which means that the server always has a surplus of processes. This has a price in memory consumption, but should reduce

the response time, since a request does not have to wait for a process to be spawn when the request arrives.

The multi–process strategies implemented in e.g. `prefork` can be interpreted as a resource controller. Consider the case, where an Apache server with the `prefork` runs simultaneously with one CPU–bound process on a fair–scheduler system. If Apache has for example three active processes, the Apache will get 3/4 of the CPU capacity in total. Assume now, that more requests arrive, and now the Apache has, for example, 5 active processes. Then the Apache uses 5/6 of the CPU capacity. Here the Apache regulates how much CPU capacity it receives according to the demands. This control mechanism ensures a relatively stable and acceptable behavior despite its simplicity.

Modules are written and compiled in a structured manner and they are loaded into the Apache server at start–up. A more detailed description of the Apache architecture and its model structure is found in *e.g.* [Laurie and Laurie, 2002; Apache Software Foundation, 2008]. Functionality can be added to the Apache server by adding hooks into a chain of phases in the request handling procedure. Fig. 2.3 roughly illustrates how an Apache process' life progresses. At initialization the Apache runs through a number of initialization phases, where a module can create hooks to add functionality.

After initialization, the process enters the request handling circle. The request cycle is run through once for every request the process handles. In Fig. 2.3 only three phases are indicated, but these phases consist of several entries for the programmer in which to add hooks. The *post config* stage is reached when the header of the new request has been read, and is thus the first place in the request circle to add a hook. After the request has been handled, and an answer has been returned to the client, the logging phase is reached. Finally, the request has been fully served, and the process is ready to serve a new request, if the process is not forced to exit.

When exiting the request cycle, the process enters the exit phase, where the module can release resources, connections or other administrative things.

2.4 Queuing Theory

Queuing theory is a mathematical discipline with various applications, such as logistics and telecommunication. The basic assumptions are that jobs (of some kind) arrive at a processor, which can only handle a certain amount of jobs simultaneously. If the processor is unavailable, the job is

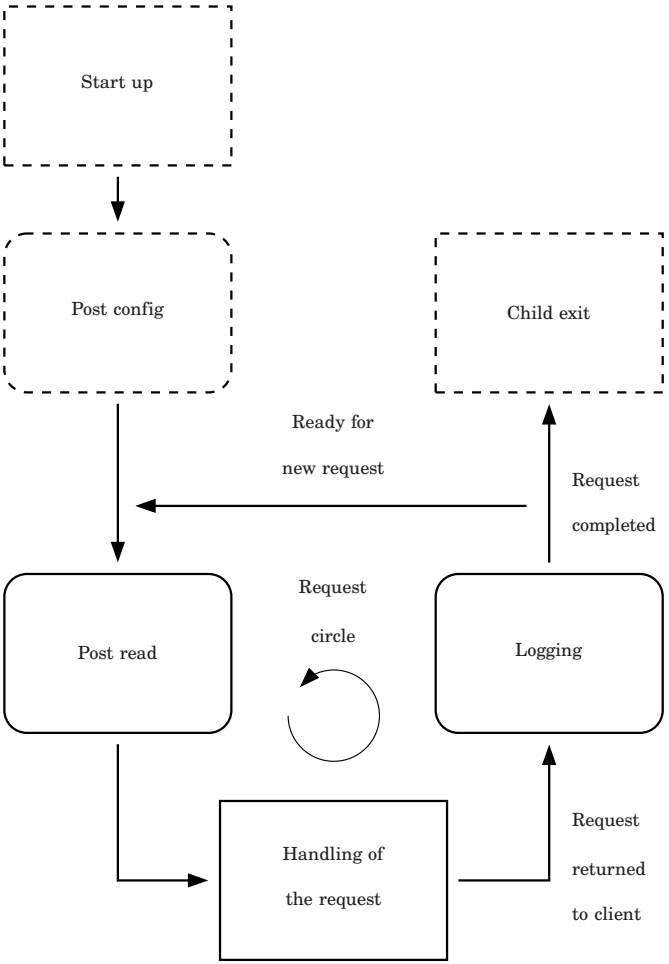

Figure 2.3 The Apache module structure. Dashed–lined boxes indicate that the phase is only utilized at the initialization and exit of the process, not necessarily activated for each request. Full–line boxes indicate that the phase is activated for each request. Round corners indicate that the module presented later adds a hook here.

either queued or rejected. By assuming some statistical properties about the arriving jobs and the time to serve the jobs, steady–state equilibrium expressions can be derived for certain metrics, such as the expected queue length and the expected time from arrival to departure. Early work of queuing theory in telecommunications was formulated in the beginning

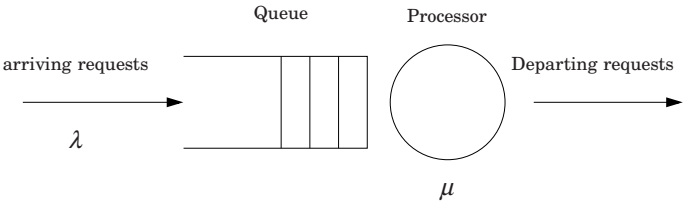

Figure 2.4 A queue.

of the 20^{th} century [Erlang, 1909]. [Kleinrock, 1975] gives a comprehensive description of the most important aspects.

Consider a queue as in Fig. 2.4. Requests arrive at a queue, where they are queued until they can start being processed. The requests arrive according to a (known) statistical distribution of the inter–arrival times with expected value λ. The processor handles the jobs according to a given scheduling strategy. The time to serve the individual requests, denoted the service time x, is given by a (known) statistical distribution with the expected value $\bar{x} = 1/\mu$, where μ is the service rate. If $\lambda > \mu$ the system is overloaded, meaning that requests are arriving faster than the processor can handle them. In this case, the system is called unstable in queuing–theory terminology. Stable operation requires that the arrival rate λ is smaller than the capacity μ.

Little's law is a kind of conservation law in queuing theory (assuming stationary conditions), which is valid for any statistical properties and any scheduler policy. It relates the response time with the expected number of jobs according to the following expression

$$d = \frac{\bar{n}}{\lambda} \qquad (\textit{Little's law})$$

In queuing theory there exists a special notation which expresses the important properties of the queue in a compact manner. The notation expresses the queue as $X/Y/K$–m. The letters X and Y are used to describe the distributions of the inter–arrival times and the service times respectively. The letter K is used to describe the amount of processors available, and the letter m can be used to describe the scheduling method. For example, $M/D/1$–PS indicates that jobs are arriving to the queue according to an exponential distribution (M for *Markovian* traffic) and the service time of the jobs are deterministic (D). The queue has one processor (1) which schedules according to the processor–sharing (PS) method (all jobs are treated simultaneously, sharing the resources equally). An $M/P/3$ queue has exponentially distributed (M) inter–arrival times and

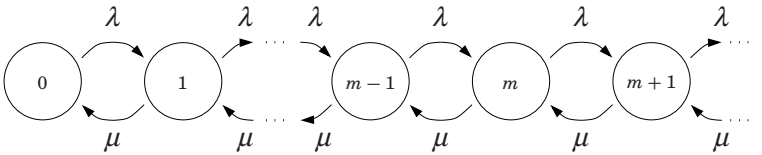

Figure 2.5 A Markov chain model of an infinite queue with a single server.

Pareto–distributed (P) service times. Three processors (3) handle the jobs according to the *first come–first served* (FCFS) method (this is the default scheduling method if nothing else is stated).

The M/M/1 queue

The $M/M/1$ queue has been studied extensively over the years because it results in fairly simple relations while it still captures the behavior of realistic queuing–systems. The assumptions are that all service times and all inter–arrival times are exponentially distributed and non–correlated. This is in many cases an over–simplification of the real–life applications, but the assumptions are often accepted because they simplify the analysis and allow for the formulation of simple relationships between the traffic parameters and the queue performance. This type of queues can be modeled by Markov chains as illustrated in Fig. 2.5. The state of the queue is the number of jobs in the system. At a given state m (except the zero state) there is an average flow of λ requests per time unit (req/s) towards the higher state $m + 1$, and an average flow of μ towards the lower state $m - 1$. Therefore, a statistical distribution for the state is given by

$$p_m = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^m \tag{2.1}$$

where p_m is the probability that the queue is in state m. Some of the nice results are the closed–form equations for the expected number of jobs in the system (both in the processor and in the queue) and the expected response time for stable systems:

- The *Expected Number of Jobs in the System* $\bar{n}$ is given by

$$\bar{n} = \sum_{j=0}^{\infty} p_j j = \frac{\lambda}{\mu - \lambda}$$

  which is an averaging of the entire queue.

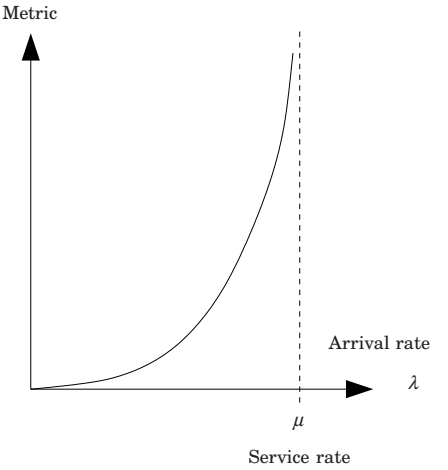

Figure 2.6 General behavior of the metrics of a queue $(\bar{n}, d)$; they grow to infinity, as the arrival rate λ is increased from zero towards the service rate μ.

- The *Expected Response Time* d is given by

$$d = \frac{\bar{n}}{\lambda} = \frac{1}{\mu - \lambda}$$

  which is found by applying *Little's law*.

These equations all show the same trend: The metrics $(\bar{n}, d)$ grow towards infinity, when λ is increased from zero towards μ, as illustrated in Fig. 2.6. This is a general behavior of a queuing system with stochastic and deterministic traffic (even with other distribution for inter–arrival times and service times).

M/G/1/PS and M/M/1 systems

Something interesting occurs when the FCFS scheduling strategy is changed to a processor sharing (PS) strategy. Assume that the requests arrive according to an exponential distribution. Then, the queuing system will behave as an $M/M/1$ queue in stationarity, no matter what statistical distribution the service time may have [Kleinrock, 1967; Noguahi and Oizurnih, 1971]. As a consequence, one does not need to consider the specific distribution of the service time, only the expected value, if the scheduling strategy is PS. This is a huge advantage since a suitable theoretical distribution for the service times can be difficult to find.

Dynamic Modeling of Queues

A dynamic flow–model of a general single–server, often denoted as *Tipper's model*, has the following form

$$\frac{d}{dt}n = \lambda(t) - \mu H\Big(n(t)\Big) \qquad (\textit{Tipper's model}) \quad (2.2)$$

where n is the dynamical state, representing the expected number of jobs in the system, and $H(\cdot)$ is the ensemble average utilization as a function of the state [Agnew, 1976; Tipper and Sundareshan, 1990; Sharma and Tipper, 1993; Wang *et al.*, 1996]. The ensemble average utilization depends on the statistical properties of the queuing system and the scheduling method. For an $M/M/1$, H is given by

$$H\Big(n(t)\Big) = \frac{n(t)}{1+n(t)} \qquad (2.3)$$

The model works well at higher loads, where the fluid–flow assumption matches the traffic well. Also, the model matches the queuing–theoretic results of Section 2.4 in steady–state.

Queuing Theory from a Control Perspective

In control terminology, traditional queuing–theory handles steady–state operation. Linearization of *Tipper's model* reveals more information on the dynamics. To allow different kinds of actuation, one often assume that the service rate (μ) can be altered in some manner by a controller. The arrival rate (λ) is considered as a disturbance, but actuation strategies, such as admission control, could transform it into an input.

The linearized *Tipper*–model, assuming an $M/M/1$ system, becomes

$$\frac{d}{dt}\Delta n(t) = \Delta\lambda(t) - \frac{\lambda^0}{n^0(n^0+1)}\Delta n(t) - \frac{n^0}{(n^0+1)}\Delta\mu(t) \qquad (2.4)$$

where the queuing–theoretic relationship

$$\mu^0 = \frac{\lambda^0}{n^0} + \lambda^0$$

has been applied. The linear model shows that the time constant T_l of the first–order system is given by

$$T_l = \frac{(n^0+1)\,n^0}{\lambda^0} \qquad (2.5)$$

This indicates that if the traffic remains the same (λ and $\bar{w}$ constant), the time constant will increase as the load is increased (n is increased by decreasing μ), and not only linearly, but quadratically. This means that the system becomes very slow if the control tries to minimize the CPU consumption. On the other hand, if the load is kept constant when the traffic parameters are increased (by also increasing μ), the dynamics of the system will become faster. This makes sense, since more jobs will go faster through the system.

The steady–state gains from μ and λ are given by

$$g_{\mu,n} = -\frac{(n^0)^2}{\lambda^0} \tag{2.6}$$

$$g_{\lambda,n} = \frac{(n^0+1)\,n^0}{\lambda^0} \tag{2.7}$$

respectively. They indicates two interesting properties:

1. As the load increases (n^0 increases), the control authority ($g_{\mu,n}$) will increase. This is both advantageous and disadvantageous. It allows for a better control, as only small changes in the control signal can achieve a large effect. On the other hand, if the controller generates too large control signals, the system is driven to extremes very fast, resulting in bad performance.

2. As the load increases (n^0 increases), the system becomes more sensitive to the disturbances (changes in λ), since $g_{\lambda,n}$ increases quadratically. This can cause problems when the controller is optimized for maintaining low μ. On the other hand, if the load remains constant when the disturbances increase (by increasing μ), the sensitivity diminishes.

The system becomes sensitive to both the control signal and the disturbances at high loads (high n^0) and the dynamics become slower. The system becomes less sensitive to both the control signal and the disturbances as the arrival rate decreases. The system then becomes faster.

2.5 Metrics for Control

Several metrics can be formulated for web servers and for computer systems in general. In the following, two web server related metrics are discussed, queue length and response time where one is manager oriented and one is client oriented. Examples of other metrics are load, utilization, memory use, rejection probability, and many others.

Queue Length

Queue–length control is the problem of controlling the average queue–length in a system. This control–approach has the clear benefit that the control output is located at the same place as the controller. Considering that the physical queue often is limited, the queue length becomes a potential control–metric. By ensuring that the system operates around a reference well below the physically limit the queue is not overloaded. The queue length as a control metric is strictly management oriented. The actual queue length is only of concern to the service manager, whereas the client does not care for the length of the queue, but rather for the service which is delivered (for example the service time). However, the queue length is related to the more client–oriented response–time (many jobs in the queue are often related to longer response times), but usually only steady–state queuing–theoretic relationships are taken into account.

In a control–context, the queue–length metric has the benefit that dynamical models for its behavior exist (*e.g.*, *Tipper's model* as in Eq. (2.2)). Examples of control systems with the queue–length as primary metric are [Kuri and Kumar, 1995; Kihl, 1999; Kihl *et al.*, 2007].

Response Time

The response time is a natural metric for control, since it is one of the most important factors for the user. Measuring the response time is rarely feasible for several reasons. First of all, it is normally not feasible for a server (where the control algorithm is implemented) to measure anything at the clients. Alternatively, the server can measure the response time locally, but the network delay is not encountered. The network delay can be estimated (*e.g.*by measuring the round–trip times of the acknowledgments in the TCP–communications). Secondly, assuming that the server could perform measurements at the clients, the next problem is to define when a request is generated and when it is served. Is a request to be defined when the user presses the button to request a URL? Is it to be defined when the TCP layer requests a connection? Is it defined when the first (or last) IP datagram is sent from the client? Similar questions can be asked regarding when a request has been served. In practice, the measurement of the response time holds many complications. Furthermore, the work on the modeling of the dynamics of the response time is very limited. Examples on response–time control are [Lu *et al.*, 2001; Henriksson *et al.*, 2004; Liu *et al.*, 2006].

2.6 Actuation Strategies

Many actuators are present in computer systems. Not all are originally intended for active control and need smaller modifications to be used. Some actuation methods are developed explicitly from the desire to change the behavior of the system by means of active control. The following presents two actuation methods relevant for control of web servers, admission control and content adaptation. Examples of other methods are the number of clients allowed simultaneously and the length of different time–outs [Hellerstein *et al.*, 2005].

Admission Control

Admission control is an actuation strategy, where arriving jobs are blocked to some degree by the controller, and thus lightens the load of the system. This actuation strategy is clearly nonlinear, since jobs can only be rejected—they cannot be invented to fulfill the demands of the controller. The admission control can be implemented in many different ways. *Token Bucket* generates tokens at a rate given by the controller. When a request arrives, it is accepted if there is a token available. *Dynamic window* allows a certain number of jobs to be present at a given time. When a request arrives, it is accepted if there is room for it, and rejected if not. Both of these strategies have the properties that in overloaded situations, the amount of jobs accepted does not depend on the arriving traffic. Another strategy is the *percent–blocking*–method, where a request is rejected with a probability set by the controller. This has the advantage that the accepted requests have the same type of statistical distribution as the arriving requests, and this simplifies the analysis. However, in overloaded situations, the number of accepted requests depends (proportionally) on the number of arriving jobs, which means that a certain increase in the arrival rate can lead to overload.

Examples of admission control are given in [Lee *et al.*, 2004; Liu *et al.*, 2006; Kihl *et al.*, 2007]). Admission control should be used with care, as denying requests is generally undesirable. Rejected requests mean loss of revenue. Admission control is therefore often used as overload protection to avoid stagnation at ultrahigh work–loads.

Content Adaptation

An alternative to admission control to handle overload is content adaptation. Here, no requests are denied, but instead the content of the reply is changed with the current load of the system. For instance, if network–bandwidth is the bottleneck, files can be better compressed or simply reduced during high load. In the case where the CPU–capacity is the bottleneck, the content of the dynamically generated pages can be altered

to reduce the required computations. Examples of content adaptation are presented in [Abdelzaher and Bhatti, 1999; Andersson, 2007].

This actuation type is not trivial to implement, and may not be relevant in many cases, since it requires that the requested content can be represented at different levels.

Virtualization

Some virtualization techniques can be used as actuators if they allow for online–adjustment of resources. Examples have been shown in the literature using Xen [Xu *et al.*, 2006; Wang *et al.*, 2007], but also other virtualization techniques have been used, see *e.g.* [Henriksson *et al.*, 2004].

Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) is an actuation method where the voltage over the CPU is altered to reduce or increase the frequency of the CPU. The main reason for this is to save electrical power and the fact that the power scales quadratically to the voltage. Therefore, the motivation to reduce the CPU frequency becomes obvious. Examples of DVS–actuation implementations can be seen in [Sharma *et al.*, 2003; Heo *et al.*, 2007].

From a control point of view, this actuation method resembles the CPU–virtualization method, as they both operate on the service time of the requests. Therefore, any control method developed for one of the actuators can easily be used on any other method. The control objective may change, however. For instance, the quadratic cost obtained from the voltage–power relationship associated with the DVS method may not be relevant for virtualization problems.

Load Balancing

Load balancing optimizes the load in a distributed system or on computer systems with several CPUs. If a unit is highly loaded (in some metric), jobs or tasks are moved to other, less loaded, units. A trade–off takes place when jobs or tasks are moved, in particular if new servers need to start up first. Because the relocation requires resources, too much relocation will increase the overhead, see *e.g.* [Kremien and Kramer, 1992; Fu *et al.*, 2006; Heo *et al.*, 2007]

2.7 Related Areas of Research

Control–theoretic methods are, of course, also applied to other areas of computer systems. In this section, a couple of examples are described in some details to illustrate the attempts to bring formalized control design methods and stability analysis into the computer–engineering world.

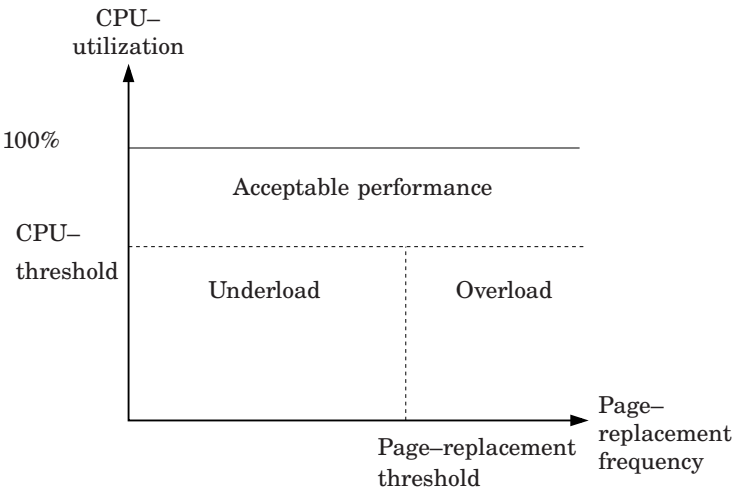

Figure 2.7 Partition of the state space into three regions; overload, underload, and acceptable operation, see [Shils, 1968], pp. 8.

An Early Load–Control Design

The first example is not directly coupled to stability analysis and formal control design, but rather serves as an example of early use of feedback in a computer system. In 1968 A. J. Shils presented a solution to a load problem observed when using the IBM M44/44X computer system [Shils, 1968]. A number of 16 users should share the common CPU and memory resources. At high load, the time to process the jobs increased significantly, despite that the CPU was not fully utilized. Paging was the root of the problem, blocking the jobs while waiting for access to the memory. This effect was self–sustaining, in the respect that arriving jobs would experience a high probability to page if the system was over–loaded, and thus only increase the problem further. A solution was to remove jobs temporally to avoid too much paging, and thus optimize the utilization of the CPU. The load was characterized by the two variables CPU utilization, and page–replacement frequency as illustrated in Fig. 2.7. The philosophy was that if the page–replacement frequency was high while the CPU–utilization was low, the system was over–loaded. If the CPU–utilization was high, the system was considered to be running smoothly. In the situation where both the page–replacement frequency and the CPU–utilization were low, the system was considered to be underloaded, meaning that less work was requested than available.

If the system was observed to be overloaded, jobs were taken out of the running–queue, and put aside in a separate waiting–queue, to reduce the

load temporarily. When the system was running in either the normal mode or in the underloaded situation, jobs were taken from the waiting–queue (if there were any), and put in the running–queue. This simple algorithm improved the throughput (number of jobs served in a given time period) during high load situations, and decreased the probability for the system to be in the overloaded and underloaded situation with 42% and 12%, respectively. The throughput was increased, but the response times were also smoothed by the control algorithm.

This work is considered as one of the first reported results on feedback control of computer systems.

Congestion Control in the Internet

The Internet consists of many local networks, and data may be transferred through several networks from one computer to another. These networks can have different capacities, and since the sender is not aware of the route the data will take, the sender cannot predict how much data the given connection can handle. If too much data is pushed through a low–capacity segment of the Internet, congestion can appear. Congestion manifests itself as long transmission times and time–outs (when the acknowledgment of a data transmission is not returned to the sender within a given time). The only solution for the sender is to reduce the amount of data sent to the network. In the Internet the TCP–layer of the network protocol–stack handles the congestion control. In the original TCP–protocol the congestion control was imposed by limiting the amount of non–acknowledged data sent to the network, determined by the co-called *window size*. The window size was static until 1988, when it was proposed to change the window size dynamically to improve the throughput and at the same time avoid congestion [Jacobson, 1988]. From this, several strategies to adapt the window size dependent on the measured time–outs have been proposed, such as *Tahoe*, *Reno*, and *Vegas*. Some improvements were based on other probing behavior, while others used different methods to detect or predict congestion based on time–outs. A bit simplified, the idea is to slowly increase the data rate offered to the network to probe the network to find the maximum data–rate possible. When congestion is detected, the data–rate is decreased dramatically to avoid further congestion, and then slowly increased again. The real implementation is a bit more complicated, including both exponential and linear increases, but the principle remains the same: Slow increase, fast decrease.

Active queue management is a strategy where datagrams are dropped from queues in the network, even if the specific queue is not congested. This helps the source to better estimate the level of congestion, and thus allows a more active congestion control. Examples of queuing policies used for active queue management are *DropTail* and *Random Early Detection*

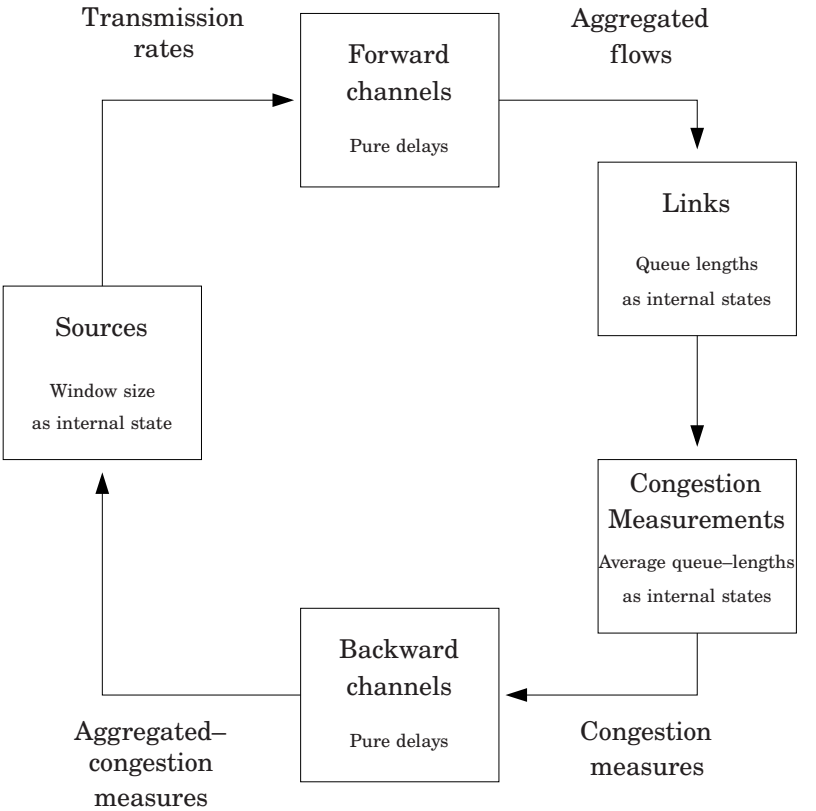

Figure 2.8 Block diagram of the congestion–control mechanism in the Internet (TCP/RED).

(RED). A more detailed description of the TCP congestion–control mechanism is presented in [Tanenbaum, 1996], while a good introduction to the development of the congestion control is presented in [Low *et al.*, 2002].

The TCP congestion algorithms have been studied from a dynamical point of view by researchers from the control community. The following describes some results presented by [Paganini *et al.*, 2001; Low *et al.*, 2003]. This group modeled the flow with TCP/RED congestion algorithm by studying flow–models. The system was divided into several inter–connected parts, as illustrated in Fig. 2.8. The sources offer a certain transmission rate to the network. The channels route these transmission rates to the individual links, modeled by routing tables and delays. Depending on the capacity of the individual links, the links respond with

a measurement of its level of congestion. In the case of RED the metric is a probability of datagram drop. This congestion information is aggregated back to the source through the backward channels, which are modeled as pure delays similar to those of the forward channels. Based on the aggregated congestion measurements, the congestion–control algorithm adjusts the window–size, and thereby the transmission rate. The congestion control–algorithm of *Reno* is modeled as the fraction between the window–size (which changes over time), and the round–trip time. The flow model of the window–size–adjustment algorithm is rather complicated, and it is based on the actual flow rate of the source and the measured congestion level. The links are modeled as queues, of which the queue lengths grow or decrease if the aggregated flow into a specific link exceeds or goes below the capacity of the link, respectively. The congestion measurement at a particular link is based on an average of the queue length, which imposes filtering.

The models obtained are non–linear and include time delays, something that makes stability analysis hard. Linearization of the models yields some interesting problems. The TCP/RED congestion control scheme becomes unstable when the delays of the network increase and also when the capacity increases. The first problem is somehow expected from a control–theoretical perspective, since it follows a general behavior for time–delayed systems. The second problem is somehow unintuitive, since one would expect problems when the capacity is too low—not when it is high. The problem arises because high capacity of the links will give higher gains in the open–loop transfer–function, which leads to instability. This causes a potential problem, since the capacity of the network is expected to increase as technology and demand evolve. This motivates for a different congestion–control algorithm, which remains stable for any capacity. Also, robustness towards transmission delays is desired, since the Internet is expected to increase, thus possibly increasing the round–trip time due to an increase in the number of links through which the flow has to pass. Following the analysis described above, new congestion–control schemes that assure local stability for any capacity and any round–trip time have been presented,. A problem with these methods is that they are based on new definitions of the congestion metrics, which require new measurements. Remember that the system is highly decentralized and it is not easy to obtain link–related measurements at the source where the controller is implemented. This often requires the datagrams to carry some extra information (some bit set in the IP header), which is altered by both the sources and the links, and retransmitted with the acknowledgments for the source to estimate the level of congestion. Furthermore, the whole analysis is based on known and constant time–delays in the network. In reality these delays will have to be estimated and will prob-

ably be time–dependent, something which is not covered by the stability analysis.

This example shows that control–theoretical methods can be used to analyze and improve the performance and robustness of a computer–related system. The major problem here is the modeling and the problem of implementing proper measurements for feedback.

3

Target System and Testbed

In this chapter a specific target–system is presented, which will be used in most of the remaining of this thesis. A testbed has been constructed to test different control strategies according to the defined target system, and this testbed is presented. The system description and the testbed are also presented in [Kjær et al., 2009].

The computer systems, which are considered in most of this thesis, host web applications that are widely used on the Internet. These types of systems can consume considerable amounts of resources, even when the system operates under normal conditions. Both from an operational point of view and from an environmental one, it is desirable to adapt the usage of resources during the changing of loads.

The system described here, along with the testbed presented, constitute the basis for the results to be presented in Chapters 4-7.

3.1 Target System

The main target system in this thesis is a general distributed computer–system hosting various web applications. Two examples of such systems are web hotels hosting several web sites, and enterprise data–centers containing business critical applications. Similar systems have been investigated in for example [Horvath *et al.*, 2007; Elnozahy *et al.*, 2003; Heo *et al.*,

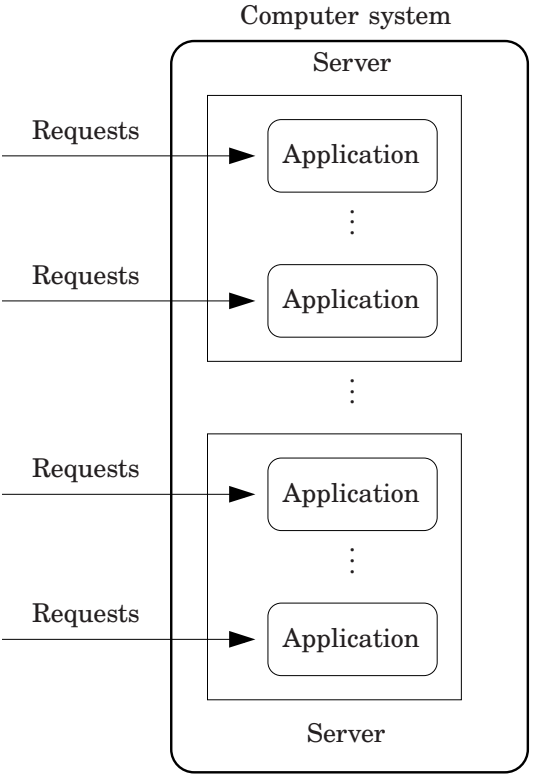

Figure 3.1 A virtualized server environment hosting web applications.

2007]. These systems are often multi-tiered. However, in the analysis one of the tiers is usually seen as the bottleneck, and thereby the analysis can be reduced to that single tier. The work presented here takes a similar approach, and the focus is on the CPUintensive tier, which processes dynamic application scripts.

The target system is shown in Fig. 3.1. New requests will arrive according to some stochastic process that may change over time. Each request can be treated independently of other requests. The physical resource of the computer system, in this case the CPU capacity, is shared among the applications using a virtualized server environment. The required work of the request, w, is a representation of the amount of work a request needs from the CPU to be processed. The required work is defined entirely by the nature of the request and cannot be affected by the control design. In this thesis the required work is measured in seconds, but it could just as well be measured in clock cycles, as for example in [Horvath *et al.*, 2007].

Each application has a *Service-Level Agreement* (SLA), defining the *Quality of Service* (QoS) that the application is guaranteed from the operator who is managing the computer system. Clients send requests to be processed by the application. Each request requires some resource capacity (here, CPU capacity) from the physical system. To fulfill the SLA, each application is guaranteed a certain share of the total CPU–capacity. Since the traffic situation may change over time, the CPU–allocation mechanism should be dynamic using some optimization criteria.

In this thesis two general assumptions about the system are made. The same assumptions have also been used in other work, for example in [Heo *et al.*, 2007; Wang *et al.*, 2007; Horvath *et al.*, 2007].

- The first assumption is that there is a load balancing mechanism, which distributes the workload among the physical servers. Therefore, all servers behave equally and independently of each other, which means that the CPU–allocation mechanism can operate on only one server.
- The second assumption is that the total CPU–capacity is large enough to respect the demand of each of the applications. With this assumption, resource allocation and management will be the focus rather than overload control. Also, with this assumption the resource allocation of each application can be controlled independently of other applications using the virtualized server environment.

As the load on the different applications can change at any time, a perfect load balancing can not be guaranteed. The implemented work–load mechanism varies from system to system. It is therefore hard to include imperfect load balancing into the work of this thesis, as the degree of imperfectness is unpredictable. The second assumption is imposed to separate the over–load problem from the resource optimization problem. If the amount of resources are not sufficient, some applications will have to reject some work to lower the load. How to divide resources optimally between several applications is an interesting and challenging problem, which has gained interest over the last years. However, the work presented in this thesis focuses on the dynamic aspects

With the two assumptions in mind only one server and one application are used in the analysis in the remaining of this thesis.

An application will have a reserved share p_r ($0<p_r<1$) of the total CPU–capacity. Non–allocated CPU–capacity, $1 - p_r$, is considered as profit–generating, since the spare CPU–capacity can be used for other purposes, such as secondary tasks (not further specified) or to save electric power by DVS. Therefore, the work presented in this thesis has the same control objective as presented in several other papers, for example

in [Wang *et al.*, 2007; Lu *et al.*, 2003; Henriksson *et al.*, 2004]; to minimize the amount of CPU capacity that is given to each application to save running costs, at the same time as the SLAs for all applications are fulfilled.

The SLAs contain the average response time for each request, meaning that an application should have a sufficient share of the CPU capacity so that its clients experience an acceptable response time from the system. In a more sophisticated SLA one could include a cost for the system operator, if the variance of the response times is too high. However, since focus here is on the technical aspects of the system rather than on the business aspects, the SLA design will not be investigated or discussed any further.

3.2 Metrics for Quality

Three metrics are chosen to show the behavior of the system. None of them are able to describe the total quality of the system, so an acceptable performance of the system yields a trade–off between several metrics.

The *average response time*, d, is a prime metric for the end user, and thus also for the application operator. For the user this metric should preferably be as small as possible, but for the computer–system operator it should be balanced with the cost of running the computer system. This balance is defined in the SLA, which is translated into control terminology as the response time reference d_r. The variable d can be regarded as the response time of a single request or as an average over a sample interval.

The *variation cost* of the response time, V_d, is a metric for how individual clients are affected by the computer system. If V_d is large, some clients will experience large response times, which is undesirable. Therefore, a low V_d is preferable, even though it is not stated as a specific SLA. V_d is defined for a stationary sequence by

$$V_d = \frac{1}{m}\sum_{j=1}^{m}(\bar{d} - d_j)^2 \quad , \quad \bar{d} = \frac{1}{m}\sum_{j=1}^{m} d_j$$

where m is sufficiently large, and d_j is the response time corresponding to the discrete–time index j. V_d is only used for long steady–state scenarios, where the system can be considered as stationary.

The *Loss of capacity*, q, is the difference between the reserved CPU capacity, p_r, and the CPU capacity actually used by the application, p_a. Since the system is sampled, p_r is constant during each sample. This metric is relevant for the computer–system operator, since it represents an operational cost, which does not generate any income. It is of high interest to keep this metric to a minimum.

For steady state cases, these three metrics are evaluated as time–averages over sufficiently long, possibly down–sampled, sequences, ensuring that the 95% confidence interval for the response time does not exceed 10% of the mean value, and that the size of the 95% confidence interval for q does not exceed 0.01 (i.e, 1% of the CPU capacity). The confidence intervals are measures of the accuracy of the average values, compared to the real expected values according to standard statistical methods, see among others [Anderson *et al.*, 1998].

Averaging over long time sequences cannot be used to improve the accuracy for transient experiments. Instead, several similar experiments are averaged to remove statistical fluctuations:

$$J_d(m) = \frac{1}{m}\sum_{j=1}^{M}\frac{1}{t_t}\sum_{l}(d_r - d_{l,j})^2 \tag{3.1}$$

$$J_q(m) = \frac{1}{M}\sum_{j=1}^{M}\frac{1}{t_t}\sum_{l}(q_{l,j})^2 \tag{3.2}$$

The above expressions represent averages over m experiments over the transient period t_t. The variables $d_{l,j}$ and $q_{l,j}$ represent the average response–time and the average loss of capacity for the l^{th} sample incident and the j^{th} experiment.

3.3 Control–Theoretic Description

The response time d is seen as the output. The actuation method (control input) is the amount of CPU capacity reserved to the server, p_r. The number of jobs in the system n is an internal state, which is often measurable. Two types of disturbances are of special concern:

Arrival rate: The arrival rate of requests to the server is a variable which is considered to be independent of the operation of the server. This is a simplification, since large response times, caused by a high load on the server, will make the users discard the web site, and thereby the arrival rate to the server is reduced. However, this situation is not covered in this thesis, as the second assumption on page 45 states that the resources are surplus. The arrival rate seen as a disturbance can be extremely important for the web server operation, since it can change very fast, as described by *e.g.* [Andersson, 2007].

Required work: This variable describes what type of resources the client is requesting. If a computationally expensive, dynamically generated, page is requested, the required work is high, whereas a static HTML file does not require much computation, and thus has a low required work. The average required work can change over time, as the popularity of the files of the site changes. If the requests shift from mainly requesting static HTML files to requesting computationally expensive, dynamically–generated, files, the average required work will increase, even if the number of requests has not changed. The required work is measured as the time it would take the server to complete the request, if this was the only task in the system. The required work is thus not only dependent on the nature of the clients, but also on the maximum capacity of the server. An alternative definition of the required work is to measure it in terms of clock cycles, as in [Horvath *et al.*, 2007], but for a given server the two definitions are equivalent, only expressed in different units. In this thesis the required work is measured in terms of time because it seems more intuitive.

These two disturbances described above are related to the behavior of the clients, a behavior which is more or less unpredictable. Using the terms *disturbance* and *unpredictable* about the behavior of the clients (often the customers), is purely associated with the terminology of the control community (just to avoid confusion with the telecommunication community).

Other types of disturbances, like the processes of the operating system and administrative tasks, can be neglected in comparison to the two main disturbances.

The requirement related to the response time in the SLA is translated into control–theoretic terminology as a reference to the response time. The SLA usually defines a fixed value, which has to be met despite the behavior of the surroundings. This means that the reference d_r can be considered to be constant, and the control objective is thus to reject the disturbances rather than reference tracking (servo problem).

3.4 Laboratory Description

A laboratory testbed has been designed to test and evaluate different web–server scenarios. The main focus was to evaluate different control structures rather than testing complex system scenarios involving database servers, application servers, multi–tier, or others. To minimize the complexity the server was implemented as a single web–server on a single

Figure 3.2 The laboratory.

standard PC, which was sufficient to represent the system described earlier in this chapter. A number of standard PCs were available for traffic generation, and the whole system was connected by standard network components. The work in this thesis investigates scenarios where the CPU causes the bottleneck, and the network was therefore dimensioned in order not to cause bottlenecks.

The setup consisted of one server, 12 client computers, and one master computer to administrate the experiments. The clients were connected to the server by an 100 Mb Ethernet switch. The master computer was connected to the lab-network through a local Ethernet network, see Figs. 3.3 and Fig. 3.2.

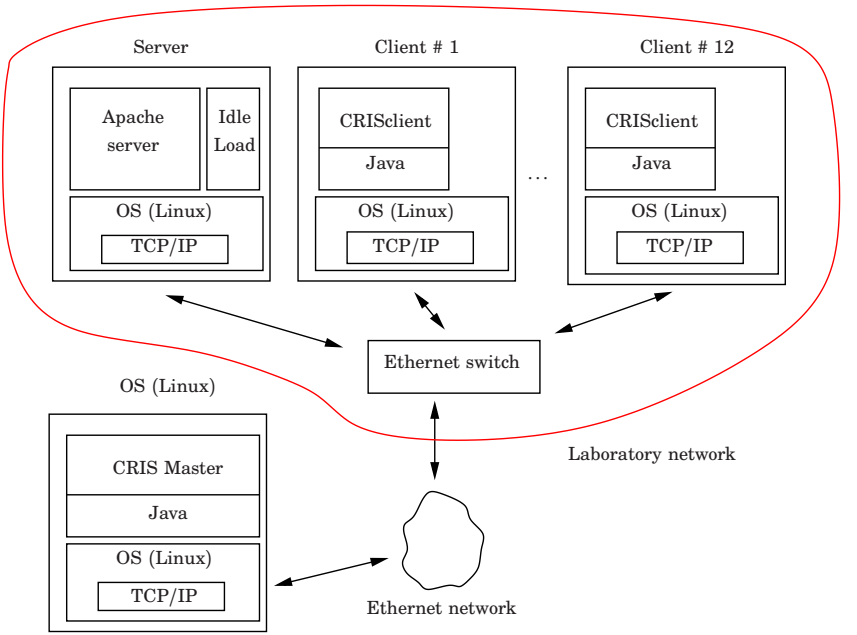

Figure 3.3 Linux based testbed with a web server, client computers, and Ethernet network.

Specifications:

- *Server*
  A Pentium 4, 1 GB memory, 3 GHz PC, with a Linux Fedora 8 operating system and modified kernel 2.6.25.4 (the Control Group functionality was included with group scheduling and accounting functionality) was used as server computer. An Apache server, version 2.2.8/`prefork`, was installed.

- *Clients*
  12 client computers, Athlon, 1.5 GHz PC with 2 GB memory, Linux Fedora 9 and kernel 2.6.26.3-29. The traffic was generated using the traffic generation software CRIS [Hagsten and Neis, 2006], controlled from the master computer.

The network delay was estimated by sending IPdatagrams from the clients to the server, and measuring the time until they were replied. 970 datagrams were sent from a client by the `ping` program under Linux, which recorded an average network round–trip–time of 266 ms.

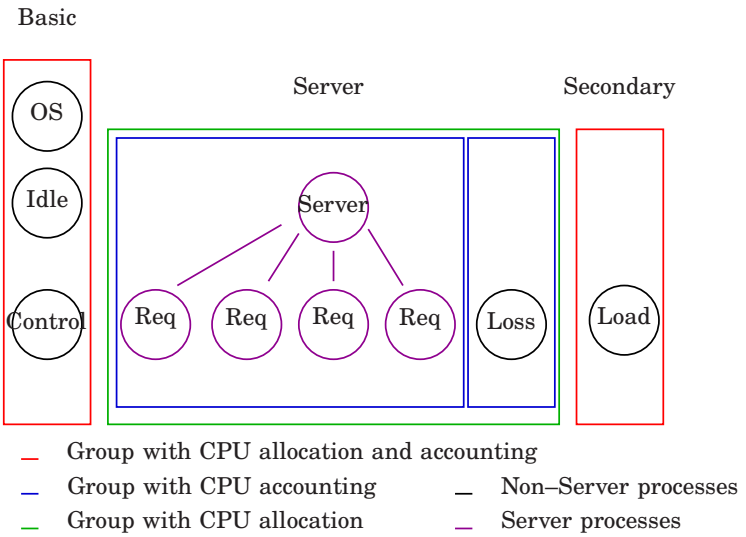

Figure 3.4 Schematic diagram of the desired grouping of processes for the laboratory implementation.

3.5 Virtualization Design

A virtualized environment was designed as actuator for the controllers. The virtualization design had to allow the controller to alter the amount of CPU capacity reserved to the web server. To ensure the operation of the computer system, the operating system had to be guaranteed a certain amount of the CPU capacity as well. The remaining part of the CPU capacity was given to unspecified applications. Measurements of CPU capacity consumptions should also be available.

Three groups were defined for CPU allocation and accounting; a group, the basic group, for operating system and other administrative tasks, another group for the server, the server group, and finally a third group, the secondary group, for secondary unspecified tasks, as illustrated in Fig. 3.4.

It was desired that certain amounts of CPU capacity could be assigned to the basic group p_r^{basic} and to the server group p_r^{server}. The remaining CPU capacity

$$p_r^{secondary} = 1 - p_r^{basic} - p_r^{server}$$

was allocated to the secondary group. In the case where the assignments to the basic group and the server group exceeded the available CPU–capacity, the basic group had the highest priority, so that the server group

was allocated what was not assigned to the basic group.

The CPU requirement for a server group was highly dependent on the clients requests. If no requests were sent to the server, the server would not consume any CPU capacity, even if capacity was reserved to the server. The amount of CPU capacity consumed by the server group was denoted p_a. The reserved, but not utilized, capacity was considered as a loss, and therefore an important metric, given by

$$q = p_r^{server} - p_a$$

To measure q, a loss process was added to the server group. Note that it was associated with its own CPU accounting group, in order for the loss of CPU capacity to be measured.

3.6 Virtualization Implementation

The focus of this thesis is actuation by changing the CPU capacity dedicated to the application. Resources such as memory, I/O are not considered, and thus, true virtual systems, such as Xen or VMware, are not necessary and would only cause overhead—both in terms of implementation and in terms of resource consumption during operation. Instead Control Groups yields a simpler solution, allowing virtualization and monitoring of the CPU capacity. Because the Apache/`prefork` configuration is so widely used, this configuration was also chosen here.

The Apache server was grouped with an idle process implemented as an infinite while–loop in a CPU–allocation CGroup. The idle process, in the following denoted loss–idle process, represented the loss of allocated CPU–capacity, as it used all capacity allocated but not used by the Apache web–server.

To distinguish between the capacity used by the Apache server and the loss–idle process, these two were placed in separate accounting CGroups.

An accounting CGroup and a CPU allocation CGroup were defined for the secondary applications on the server system, see Fig. 3.5. These applications were assumed to use the capacity that was not used by the target application, implemented as an infinite while–loop.

All remaining processes (operating system processes, administrating processes, and the controller) were collected in an accounting CGroup and a CPU allocation CGroup.

The loss–idle process and the control process were implemented by using special requests to the Apache server and by moving these processes to the relevant CGroups, as indicated by the arrows.

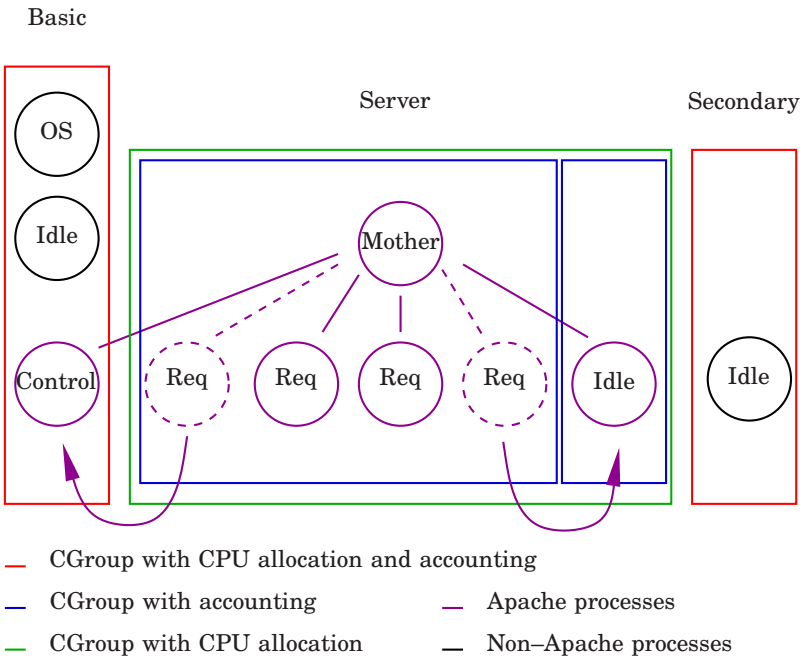

Figure 3.5 Schematic diagram of the processes and CGroup for the Apache implementation.

The loss–idle process was implemented in the normal Apache request handling sequence. When the `idle.start` file was requested, an Apache *log_transaction* hook started an infinite while–loop. This special request responded with a simple `html` answer, but never finished the logging phase. This means that the while loop used a process as long as the while loop existed. This did not cause any problems in the normal use of the server, as the Apache spawned new processes when needed (in the case of `prefork`). The while–loop was governed by a lock implemented with a semaphore. When a normal request arrived, it checked if it was the only (normal) request being served. If this was true, it locked the semaphore, and the while–loop stopped. Likewise, when a normal request finished, it checked if it would leave the system empty for normal requests, and if this was the case, it released the semaphore. When the loss–idle process was initiated, it first looked up its own process–id and then moved itself from the Apache accounting CGroup to the special loss CGroup for accounting by writing the process–id to the relevant `task`–file.

The controller relied on access to the measured variables, which were

implemented in the Apache server. Therefore, the controller was implemented inside the Apache server as a special request with similar structure as the loss–idle process. When the `periodicctrl.start` file was requested, the handling process entered an infinite loop during the logging phase. For each loop a control action (reading of relevant measurements and variables, calculation of a new control signal, and setting the relevant `share`–files) was performed, the loop slept for a specified amount of time before waking up for a new sample. Because the calculation of the control signal and the setting of the actuator did not happen instantaneously, this delay was measured and subtracted from the desired sampling interval to obtain an accurate sleep–interval. Before the control process entered the infinite loop, it moved itself to the basic CGroup, both for CPU allocation and for accounting.

The implementation required the use of three hooks into the Apache request–chain (see Fig. 2.3 on page 29).

- post_config: This hook entered the chain at a quite early stage of the process life, where initialization of the process itself took place. A shared memory area dedicated to the prediction/control functionality was implemented here to allow communication between the processes.
- post_read_request: This hook entered the request chain when the request had been defined, and here all information about incoming requests was updated. Most importantly, the number of active jobs was updated here.
- log_transaction: In this hook, the variables updated in the *post_config* phase were updated again.

To avoid the problem of parameters being updated by one process while a second process is reading them (and assuming them to be static), a locking mechanism was imposed by a semaphore. During a request cycle, the shared memory was locked and unlocked twice; when the parameters were updated in the *post_read_request* stage, and when the parameters were updated in the *log_transaction* stage.

3.7 Timing Issues and Quantification Errors

Sampling Intervals

On the scheduler level only one application had access to the CPU at a specific time, and this application utilized the CPU 100%. This was a discrete behavior. On a larger time–scale, the fraction of CPU capacity given to a

specific application could be considered as a continuous variable between 0 and 100%. The question was how small the time scale could be before the discrete behavior became dominant. This investigation was important for determining the smallest sampling interval, as the implementation should render a processor–sharing system. A simple setup was constructed to test how short sampling intervals that can be applied, while still resembling a processor–sharing system.

Three CGroups were defined. The operating system and other administrative tasks were collected in CGroup *g0*. Two infinite while–loops are placed in CGroup *g1* and CGroup *g2*. The purpose of these while loops was to consume as much CPU capacity as possible, to have smooth CPU consumptions for evaluation. To ensure that CGroup *g0* consumed the CPU capacity given to it, an infinite while–loop was added to the CGroup. The reference to group *g0* was constant at 15%, while the CPU references to the two other groups were varied.

Figure 3.6 shows how the scheduler followed a reference for different sampling intervals when reading the accounting files. All the figures show that the references were followed accurately in steady–state. Sampling periods of one second showed nice and exact steady–state reference following in the top of the figure. With sampling intervals of 100 ms some fluctuations around the reference were observed. This was an effect of the non–ideal processing–sharing, which was implemented in the scheduler. The effect was more pronounced when the sampling period was 10 ms.

From the top of Fig. 3.6 the response to a step change can be seen to be less or equal to 1 s. The averaging effect of the slow sampling may hide a faster response, and this is seen in the middle of Fig. 3.6, where the scheduler seems to have responded within 200 ms or less. Since this is in the order of the sampling period, faster response may be hidden in the averaging. The bottom of Fig. 3.6 shows that the scheduler responded after 10 ms or less. Due to the fluctuations it was not possible to investigate at lower sampling periods.

Figure 3.6 suggests that sampling intervals around 100 ms ensure that the response time of the scheduler could be neglected, and the scheduler acted close to ideal processor–sharing. Shorter sampling intervals were feasible from the scheduler–response point of view, but the fluctuations observed at the bottom of Fig. 3.6 indicate that the scheduler could no longer be regarded as processor–sharing. The investigation also suggested that the dynamics of the actuator could be neglected compared to the lower boundary of the sampling frequency at approximately 100 ms.

Actuator Dynamics

To get an indication of the delays when accessing the *share* files and the accounting files, time measurements (clock readings) were taken at ap-

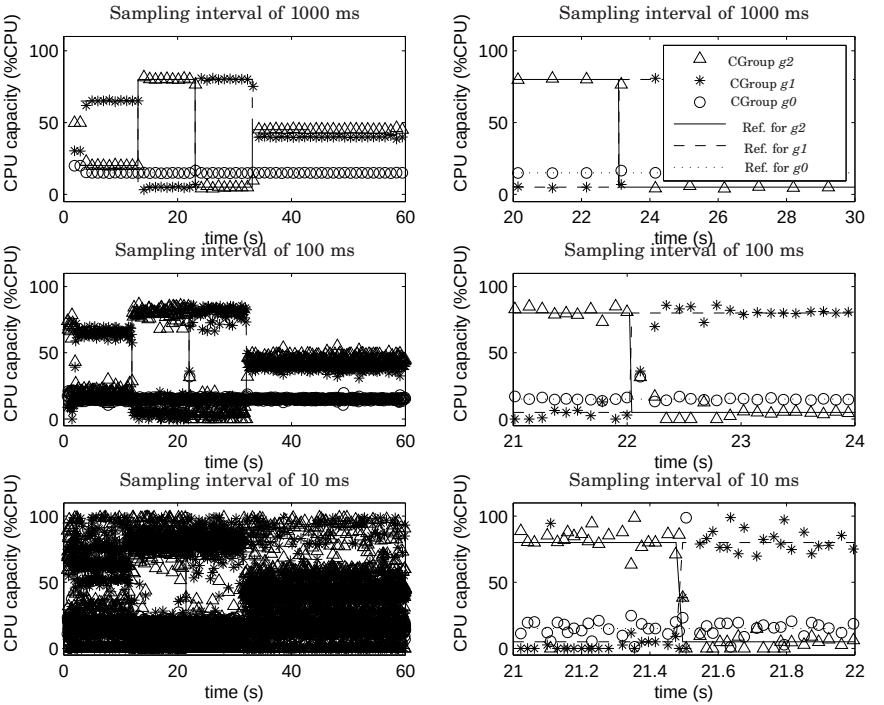

Figure 3.6 Allocation of CPU with changing references and different sampling intervals. The right–hand side is a magnification of a reference change in the left–hand side of the figure. The smoother a signal is, the better the approximation to processor sharing the system was.

propriate places in the code. Readings of the clock were performed before and after each reading and writing operation.

Figure 3.7 shows that reading seven accounting files and writing three `share`–files took in the order of 0.1 ms - 0.2 ms.

Quantification of the Control Signal

The share values were represented as integers, which means that the obtainable CPU capacity reserved to the server was quantified. To minimize the quantification problem, the maximum *share*–value was always associated to the CGroup with the highest CPU–fraction. The remaining *share*–values were then calculated to obtain the desired relative relations between the CPU fractions. Using a maximum *share*–value of 1,000,000 resulted in a maximum quantification error of 0.0002% of the maximum *share*–value (found by numerical evaluation), which was considered acceptable.

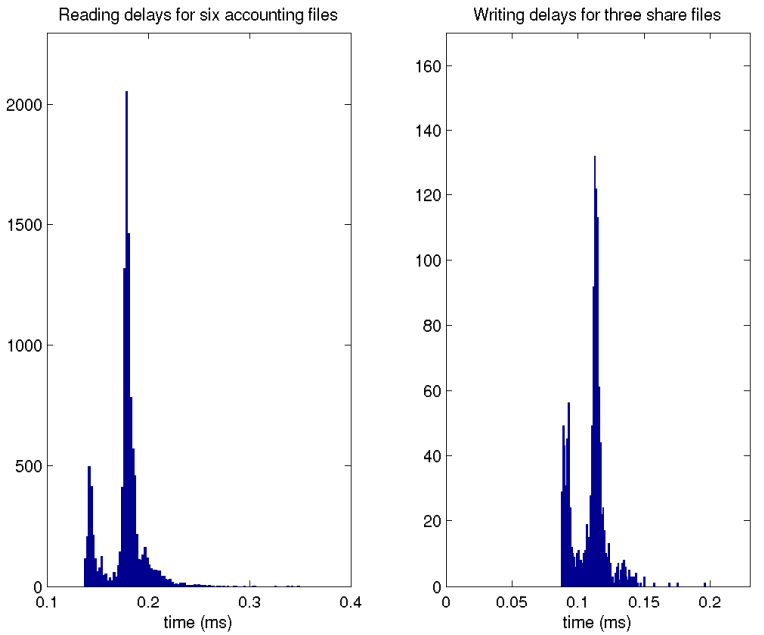

Figure 3.7 Timing histograms when accessing the share files and the accounting files.

Dynamics of the Idle Process

Figure 3.8 shows the results from a test of the idle–process implementation. A request for a single PHP document was sent to an empty server. The PHP request required 16.3 s to complete. As seen in the top of the figure, the idle–process consumed all CPU capacity dedicated to the server. When the request arrived, all dedicated capacity was used for handling the request. When the request was handled, the idle–process again consumed all the dedicated CPU–capacity, and as the server was now empty, the request–processes consumed no CPU capacity. The two bottom figures show the CPU allocation just around the arrival time and the departure time. These figures indicate (within the accuracy of the sampling time of 20 ms) that the idle–process acquired and released the dedicated CPU–capacity sufficiently fast.

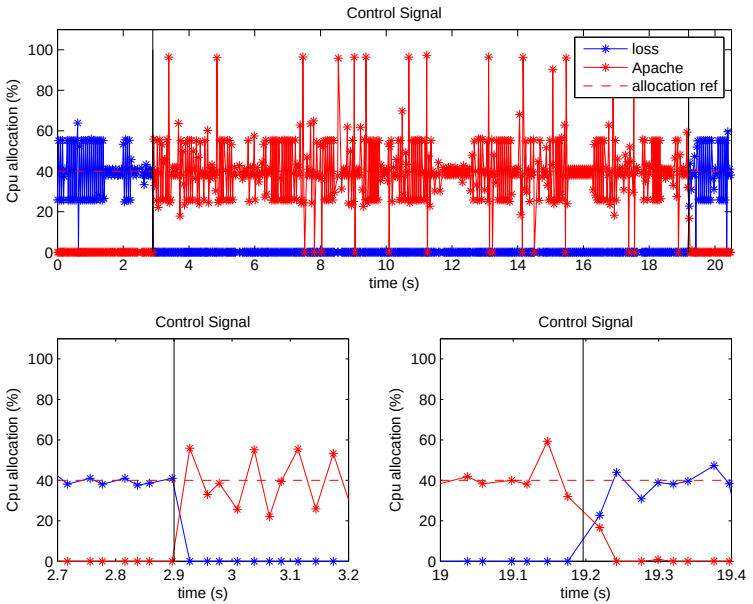

Figure 3.8 Test results for idle–process experiment. A web page was requested around time t =2.9 s. The job was completed around time t =19 s. The CPU capacity was sampled with 100 ms intervals. The CPU capacities were read with 20 ms interval from the accounting files. The black lines indicate the arrival time and the departure time, respectively.

3.8 Traffic Generation

Traffic generation for web–server experiments is a complicated issue because traffic characteristics are not the same from web site to web site. When designing traffic for a given experiment the relevance of the results can always be questioned compared to certain applications, as other applications will be exposed to other types of traffic. The traffic for the testbed is chosen to resemble that of large modern news–cites as these sites are exposed to large and highly fluctuating quantities of requests.

Different solutions are available for generating workload for web–server systems, such as RUBiS [RUBiS, 2009] and SURGE [Barford and Crovella, 1998]. In this thesis, traffic was generated with the CRIS tool, which is a java based software tool developed in a large research project related to crisis emergency management at Lund University [Hagsten and Neis, 2006]. The CRIS tool is based on real–life data traces from Sweden's largest news site. This means that both the traffic model and the request

distribution are based on real data.

CRIS allows several clients (computers) to unite to generate traffic with the specified distribution. Traffic–information files defining both the arrival times of requests and the requested documents are uploaded to the clients prior to an experiment. Therefore, the same traffic information files can be used for several experiments, providing an easy way to compare different system implementations.

For the server in the testbed, CRIS generated a number of PHP files. A PHP request then generated a string of characters the length of which was fixed for the given file, but varied over the total amount of PHP files with a predefined distribution. Also, a distribution on the document popularity was configured, which determined the probability for each PHP file to be requested.

Inter Arrival Times

The inter–arrival times were defined by distributions prior to the experiment. Fig. 3.9 shows the online measured inter–arrival time distributions compared to the desired exponential distributions. As observed from the figure, the distribution of the measured inter–arrival times corresponded well to the desired distribution.

Offline Estimation of the Average Required Work

The required work w is a key variable in this thesis, but it was not possible to measure in the testbed. The average required work $\bar{w}$ was estimated offline by profiling techniques.

The file–popularity distribution and the character distribution derived by the CRIS tool did, of course, influence the distribution of the required work. Estimates of the distribution of the required work and, in particular, an estimate of the average required work were important for analysis with queuing theory, dynamic modeling, feedback control design, and feed–forward control design. Therefore, offline experiments were conducted with low arrival rates, similar to the procedure used in [Liu *et al.*, 2006; Henriksson *et al.*, 2004].

Four experiments were conducted; two different popularity distributions (same as in later experiments) and two different values of the CPU–allocation parameter, p_r (kept constant during the experiment). The inter–arrival times were set to one second (deterministic distribution). A number of 5000 requests were used for each experiment. A necessary modeling assumption for the offline estimations was that the response time of the request, d, was only dependent of the required work, w, and the CPU–allocation parameter, p_r. This means that an estimate of the

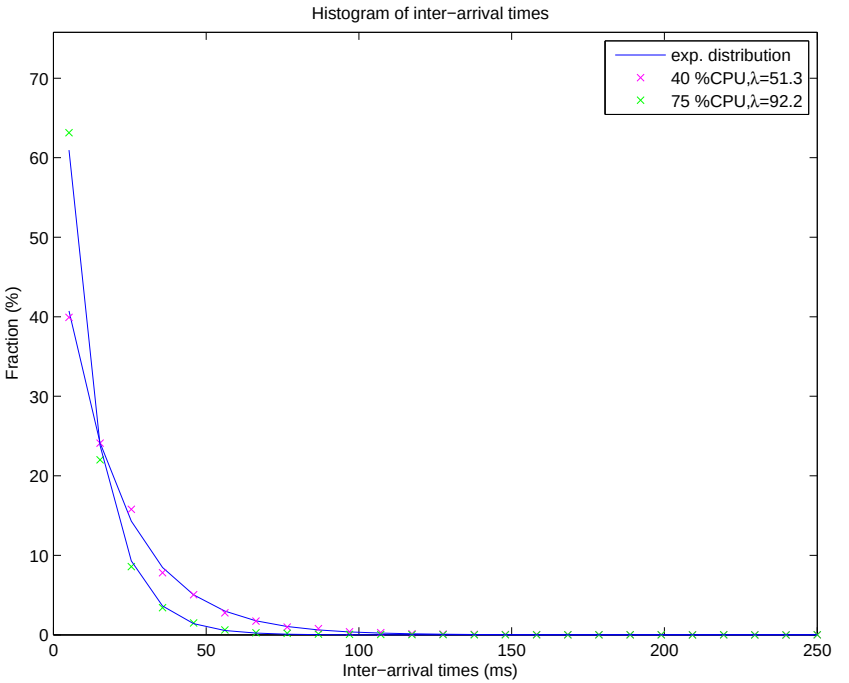

Figure 3.9 Distributions of inter–arrival times for the two arrival rates λ=50 req/s and λ=100 req/s. An entry in the figure represents the interval half way between its neighbors. For instance, the entry at inter–arrival time = 25.5 ms represents the interval from 20.4 ms to 30.6 ms.

required work for a request, $\hat{w}$, could be calculated as

$$\hat{w} = d \cdot p_r$$

The distribution of the required work is illustrated in Fig. 3.10, which shows that the distributions were not exponential.

The estimations of the average required work are listed in Table 3.2. The estimates should have been independent on the CPU– allocation parameter. However, as seen in the table, this is not the case. This result indicates that other factors than the actual CPU–processing, such as I/O handling and memory handling, affected the response time, and thus, the model is not accurate. In control theory, model errors do not necessarily yield poor performance due to the properties of feedback. However, if feed–forward is used, model errors can degrade the performance.

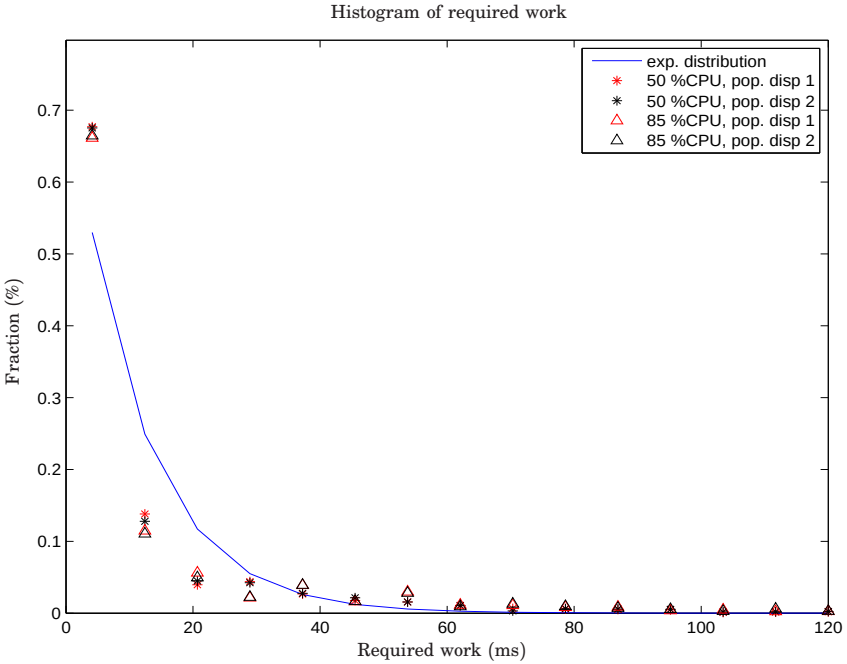

Figure 3.10 Distributions of required work for four different experiments. An entry in the figure represents the interval half way between its neighbors. For instance, the entry at required work = 20.7 ms represents the interval from 16.6 ms to 29.0 ms. Two different document popularity distributions and two different amounts of allocated CPU capacity were tested. The estimated required work was calculated by $\bar{w} = \bar{d} \cdot p_r$, where $\bar{d}$ was the average response time and p_r was the (constant) allocated CPU capacity. The full line represents an exponential distribution with the same mean as the 50% of the CPU, popularity distribution 1–experiment.

3.9 Discussions and Conclusions

The domain for the remaining of the thesis has been described in this chapter. The target system is a web server running on a computer. The CPU capacity available for the server can be changed dynamically and can be seen as the control signal. The main metric for control is the response time from when the clients request the files until the they receive the corresponding answers. The purpose of control is to maintain a constant average response–time despite disturbances. Disturbances can for example be changes in the arrival rate.

More sophisticated target systems than the one presented here could be multi–tier systems, where jobs pass through servers, databases, and

Table 3.2 Average required works, found from low–rate experiments

Popularity distribution	%CPU **50**	**85**
1	10.7 ms	14.1 ms
2	11.8 ms	14.9 ms

application servers located on several physical computers. In this thesis the focus is on the dynamic properties and disturbance rejection, and not on a larger resource distribution problem. Both problems represent important issues which have to be solved. Eventually, real implementations should incorporate them both, and also here solutions are emerging, as *e.g.* [Jayachandran *et al.*, 2009; Padala *et al.*, 2009b].

A testbed was designed to test and verify different control schemes for a target system as above. Actions (calculations, measurement and control actions) can be triggered both periodically and by departure events where a departure occurs when a request is completed and a response is sent back to the client. Results from the testbed should be not be considered as representative for real traffic on the Internet as this holds far more problems. The qualitative behavior should be comparable with the limitations described. The implementation should not be transferred directly to a real system for the Internet. The implementation presented in this chapter serves only to test the control method and does not consider safety issues, user interaction, and other system specific issues. However, the basic improvements obtained by the proposed methods and the analytical results are assumed to be general and should also give similar results if investigated with real systems.

4

Modeling

The work presented in this chapter was initiated by oscillations observed in the laboratory. The focus is therefore given to a specific problem, related to feedback and feed–forward control of the response time via virtualization. The modeling approach is similar to the one often taken in queuing systems; the stochastic nature of the traffic is approximated as a flow, which leads to continuous differential equations. The contribution here is the inclusion of external buffering which introduces dynamics previously not reflected in the measurements. The investigations demonstrate that the consequence of the extra buffering is that the measurement of a disturbance actually is state dependent. The material presented in this chapter is based on [Kjær and Robertsson, 2009; Kjær and Robertsson, 2010].

Web servers seldom run directly on the hardware of the computer. Several layers of software are implemented below the web server, often collected in the operating system. The philosophy behind the layering is that as long as the defined interfaces between the layers are respected, the functionality of one layer can be exchanged without redesigning the other layers. Examples of layers in a modern computer are the IP layer and the TCP layer. A whole framework of layering is defined in the OSI model and the TCP/IP stack, which both define a number of layers, where each layer only has knowledge of the interface to the layer above and the layer

below [Tanenbaum, 1996]. One of the methods to keep the layers independent of the behavior of the other layers, is to insert buffers. Buffers are simple queues without processors. When a layer has something to sent to the upper layer, it may put the job in its own out–going buffer, and is then ready to proceed with other tasks. When the above layer receives a job, it may put it in its own in–going buffer until it is ready for treating it. If this is done throughout the whole stack of layers, a request send to a web server may pass through several buffers before reaching the actual web–server. All this buffering optimizes throughput, since the individual layers do not have to wait for either the upper or the lower layer to respond. The drawback is that determination of the time for a request to propagate through the whole stack is not possible, and the response times are therefore out of control.

When trying to alter the dynamics of the system by means of control (trying to control for instance the average response time), the buffering will even alter the dynamics of the whole queuing system, since the propagation through the buffers is a dynamic problem. This chapter studies the effects of these underlying buffers to the dynamics of the queuing system.

4.1 Internal and External Buffers

Assume that the target problem is to control a web server running on top of a stack as described above. The number of buffers is unknown, and, naturally, it is not feasible to measure any quantities in these buffers (such as buffer length or buffering times). At least one underlying buffer is known to exist in the test bed described in Chapter 3; the `Backlog` buffer stores TCP sockets (endpoints of Internet communication flows) until a web server process becomes available. Other buffers are also likely to be present. All underlying buffers will from now on be denoted as *external buffers* to distinguish them from the web server.

It is assumed that all the buffers implemented before the web server are ideal buffers, and they do not impose any delay if the following element in the chain is willing to accept the job. The effect of other jobs passing through the same buffers is neglected, assuming that the requests to the web server are not delayed by jobs of other purposes, such as ftp and ssh. The downstream buffers, which are passed when the answer is sent to the client, are also neglected. The simplified view of a web–server system is illustrated in Fig. 4.1.

The web server is modeled as a single processor–sharing server with a limited capacity in terms of the number of jobs allowed simultaneously (the internal queue can hold a maximum of M_c requests). To simplify the analysis, it is assumed that both the inter–arrival times of the re-

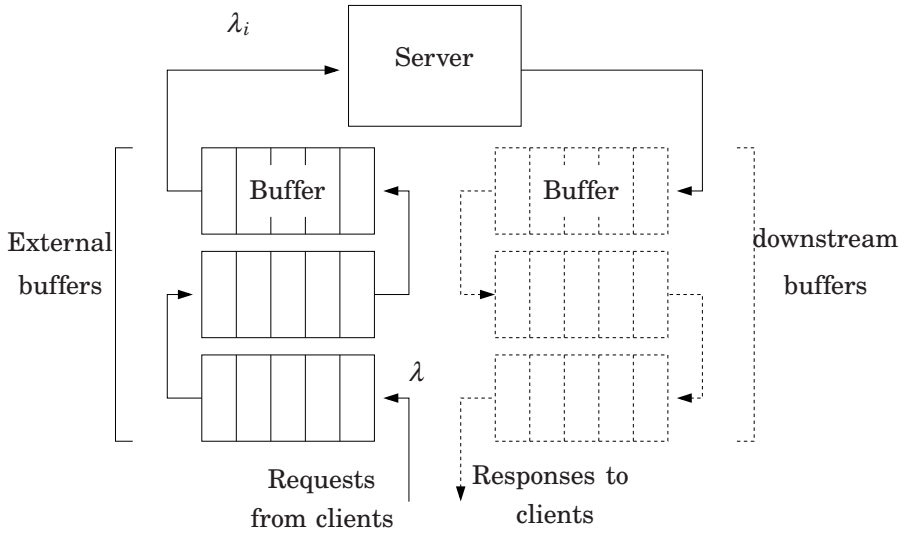

Figure 4.1 Requests may pass through several queues before reaching the server. The dashed path indicates buffers which are not considered in this work.

quests and the required work w of the requests are uncorrelated and exponentially distributed. Since the model is intended for control design and analysis, some inaccuracy can be accepted. Control theory has tools to handle model inaccuracy, and feedback control may reduce the effect of inaccurate models.

The individual external buffers do not hold any independent processors and the external buffers should therefore not be modeled as individual queuing systems, but rather as one long queue. The only processor in the system is that of the actual web–server, and the dynamics of the entire queuing system is therefore modeled as one infinite queue and a single processor as illustrated in Fig. 4.2.

4.2 Static Modeling

First, the system is considered to be in steady state and all dynamics are neglected. It is also assumed that the queue is stable, that is, $\mu > \lambda$.

The queue is modeled with a Markov chain as illustrated in Fig. 4.3, and the probability distribution for the specific number of jobs in the

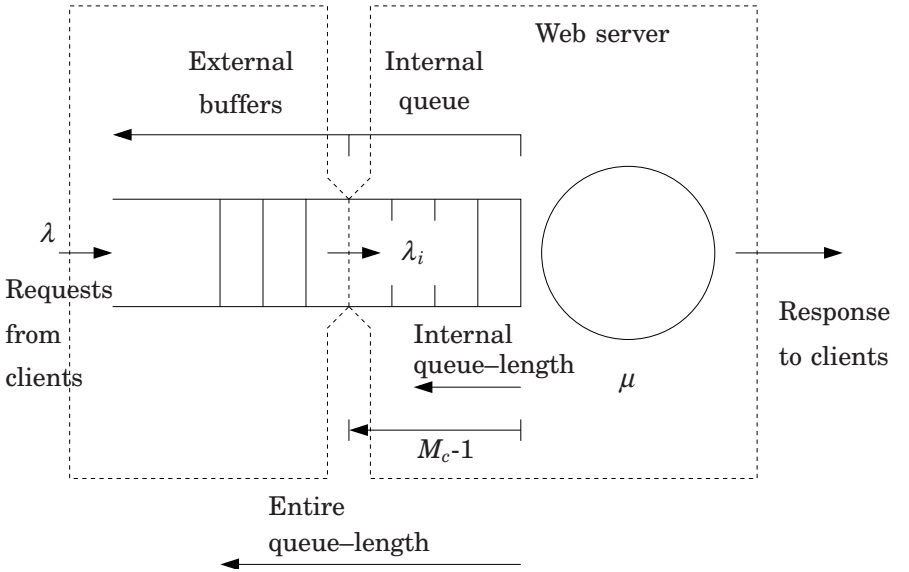

Figure 4.2 Model of the entire queuing system. All queues are gathered into one infinite queue, accepting the arrival rate λ. Only a part of the queue, the first M_c entries, is visible for the server, and the flow of requests into this part is denoted λ_i. The measured queue–length inside the server is denoted n_i.

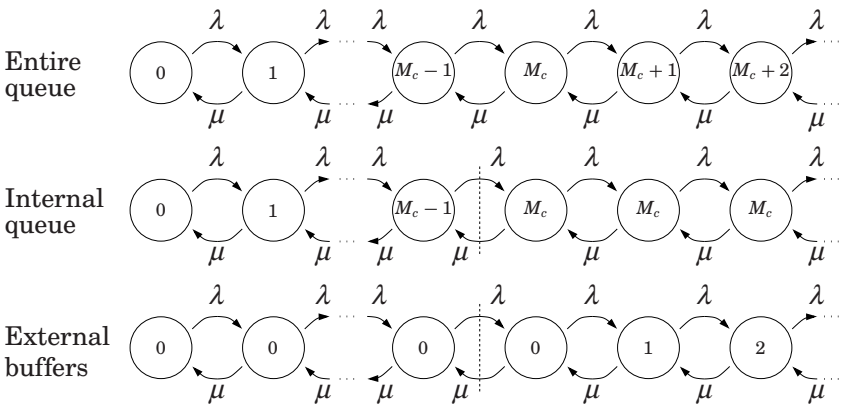

Figure 4.3 Markov chain representing the entire queuing system (*upper*), the internal queue (*middle*), and the external buffers (*lower*). The variables associated with the arrows represent the flow from one state to another, while the variables inside the circles represent the measured value at the given state.

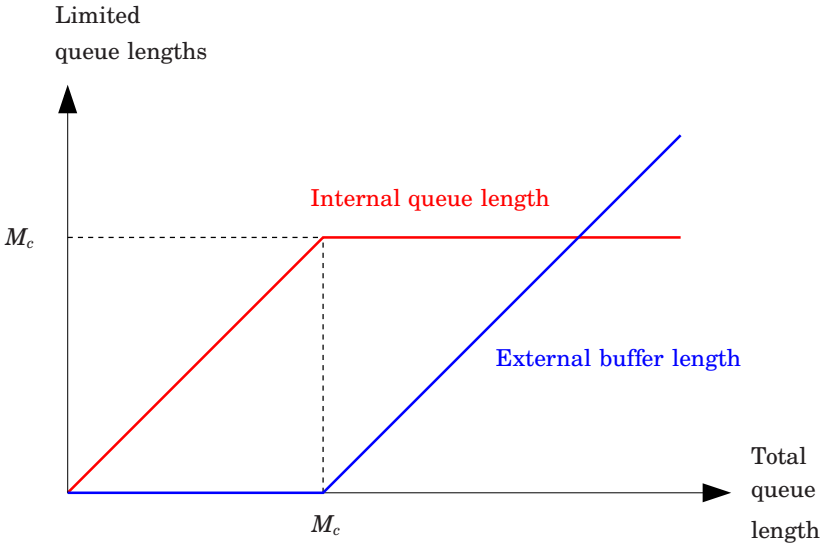

Figure 4.4 Relation between the instantaneous length of the entire queue and the instantaneous lengths of the internal queue and the external buffers.

system is given by

$$p_j = \left(1 - \frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^j \tag{4.1}$$

because it is an $M/M/1$ queue. This distribution is associated with the queue and it is therefore identical no matter which part of the queue taken into consideration.

The average length of the entire queue $\bar{n}$ is then

$$\bar{n} = \sum_{j=0}^{\infty} p_j j = \left(1 - \frac{\lambda}{\mu}\right) \sum_{j=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^j j = \frac{\frac{\lambda}{\mu}}{1 - \frac{\lambda}{\mu}} \tag{4.2}$$

which is a well–known result from queuing theory.

At a given time, the instantaneous queue–lengths of the internal queue and the external buffer are given entirely by the instantaneous queue–length of the entire queue. If the entire queue has less jobs than the limitation (M_c), the internal queue–length will be identical to the entire queue. As all jobs are in the internal queue, the external buffers are empty. If the entire queue holds more jobs than M_c, the internal queue will hold exactly M_c jobs, while the external buffers will hold the remaining

jobs. These relations are illustrated in Fig. 4.4. To evaluate the average length of the internal queue $\bar{n}_i$ and the external buffers $\bar{n}_e$, the Markov chain is divided into two parts as illustrated in Fig. 4.3. The left–hand side of the chain represents the situations where the number of jobs in the entire system does not fill up the internal queue. Here, the internal queue is treated the same way as when evaluating the entire queue. The external buffers do not hold any jobs in this case and the contribution to the average value is therefore zero. The right–hand side of the chain represents the situation where the internal queue is full and queuing in the external buffers may occur. Here, the internal queue observes a number of M_c jobs, and the external buffers observe an increasing number of jobs starting from zero. The expected values of the internal and external buffers therefore become

$$\bar{n}_i = \sum_{j=0}^{M_c-1} p_j j + \sum_{j=M_c}^{\infty} p_j M_c \tag{4.3}$$

$$\bar{n}_e = \sum_{j=0}^{M_c-1} p_j\, 0 + \sum_{l=M_c}^{\infty} p_l (l - M_c) \tag{4.4}$$

Adding the two expected queue–lengths gives

$$\bar{n}_i + \bar{n}_e = \sum_{j=0}^{M_c-1} p_j j + \sum_{l=M_c}^{\infty} p_l \,(M_c + l - M_c) = \sum_{j=0}^{\infty} p_j j \tag{4.5}$$

Comparison to Eq. (4.2) reveals an important property:

$$\bar{n}_i + \bar{n}_e = \sum_{j=0}^{\infty} p_j j = \bar{n} \tag{4.6}$$

The expected value of the entire queue–length is the sum of the two expected values of the queue lengths of the individual sub–queues. This may seem trivial, but trivialities are rare in queuing theory.

Elaborating further on the expression for $\bar{n}_e$ from Eq. (4.4) reveals

$$\bar{n}_e = \sum_{j=0}^{M_c-1} p_j\, 0 + \sum_{l=M_c}^{\infty} p_l (l - M_c) \tag{4.7}$$

$$= \left(1 - \frac{\lambda}{\mu}\right) \sum_{j=M_c}^{\infty} \left(\frac{\lambda}{\mu}\right)^j (j - M_c) \tag{4.8}$$

which is rewritten into

$$\bar{n}_e = \left(1-\frac{\lambda}{\mu}\right)\left(\frac{\lambda}{\mu}\right)^{M_c}\sum_{j=0}^{\infty}\left(\frac{\lambda}{\mu}\right)^{j} j \tag{4.9}$$

$$= \frac{\frac{\lambda}{\mu}}{1-\frac{\lambda}{\mu}}\left(\frac{\lambda}{\mu}\right)^{M_c} \tag{4.10}$$

Comparing to Eq. (4.2), a relation to the entire queue–length is given by

$$\bar{n}_e = \bar{n}\left(\frac{\lambda}{\mu}\right)^{M_c} = \bar{n}\left(\frac{\bar{n}}{1+\bar{n}}\right)^{M_c} \tag{4.11}$$

The modeling of the internal queue is simplified by combining the above equation and Eq. (4.6):

$$\bar{n}_i = \bar{n}-\bar{n}_e = \bar{n}\left(1-\left(\frac{\lambda}{\mu}\right)^{M_c}\right) \tag{4.12}$$

$$= \bar{n}\left[1-\left(\frac{\bar{n}}{1+\bar{n}}\right)^{M_c}\right] \tag{4.13}$$

or formulated as a function of the load

$$\bar{n}_i = \frac{\frac{\lambda}{\mu}}{1-\frac{\lambda}{\mu}}\left[1-\left(\frac{\lambda}{\mu}\right)^{M_c}\right] \tag{4.14}$$

Figure 4.5 illustrates the internal queue–length and the external buffer–length as functions of the total queue–length. The behavior illustrated in the figure follows the intuition: The internal queue grows with $\bar{n}$, but does never exceed M_c, which is the upper limit for the number of jobs that the web server can hold (represented by the horizontal dashed–line). For low values of $\bar{n}$ the external buffers are empty because the web server can hold all the jobs, but as $\bar{n}$ becomes higher, the external buffer lengths grow. At very high values of $\bar{n}$ the external buffers hold all the jobs except the M_c jobs which are held in the web server. What may come as a surprise is that the internal queue–length curve starts to break so early. If, for example, a measurement reads 40 req, the actual queue–length is rather in the order of 50 req. One may consider a reading of 40 req to be medium loaded, and neglect the effect of the limited queue. However, as seen in this example, this is not the case. As a consequence, the limitation on the queue length does not only have effect when the queue length is close to the limit.

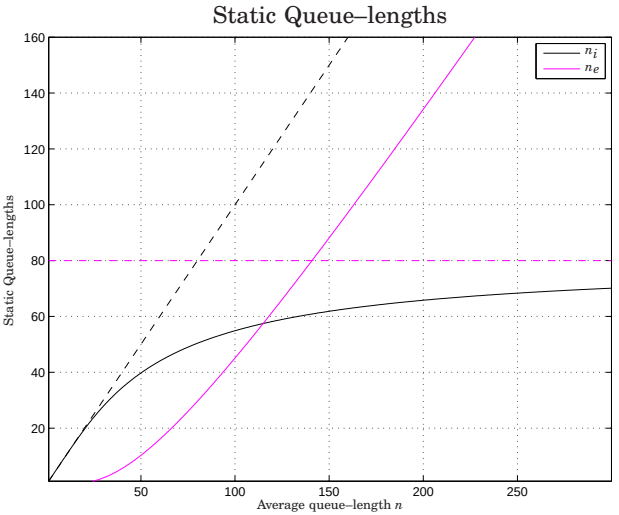

Figure 4.5 Static average queue–lengths for M_c=80. The horizontal dashed–line represents the upper limit for how many jobs the web server can hold (M_c).

As the queue is assumed to be in steady state, the flow into the external buffers must be the same as the flow out of the external buffers and thus $\lambda = \lambda_i$. The response times of the entire queue, d, and the response times of the internal queue d_i are derived from *Little's law*:

$$d = \frac{\bar{n}}{\lambda} = \frac{1}{\mu - \lambda} \tag{4.15}$$

$$d_i = \frac{\bar{n}_i}{\lambda_i} = \frac{1 - \left(\frac{\lambda}{\mu}\right)^{M_c}}{\mu - \lambda} \tag{4.16}$$

Assuming that the service time can be controlled by *e.g.* CPU virtualization or DVS, the service rate is given by

$$\mu = p_r / \bar{w} \tag{4.17}$$

where $\bar{w}$ is the average time the jobs would require to be served (not including queuing time), if only one job were processed at a time.

4.3 Dynamic Modeling

A queue length model for the entire queuing system, without taking explicit notice of what is observed inside the web server, can be described

by *Tipper's model* (see also Section 2.4):

$$\frac{d}{dt}n(t) \quad = \quad \lambda(t) - \mu\frac{n(t)}{1+n(t)} \tag{4.18}$$

assuming exponentially distributed inter–arrival times and service times. Here, the arriving traffic is entering the first external buffer, which may not be measurable in a real system. Assuming that the service rate μ can be changed online by altering the reserved CPU–capacity p_r (by some virtualization technique or by dynamic voltage scaling), the model becomes

$$\frac{d}{dt}n(t) \quad = \quad \lambda(t) - \frac{p_r(t)}{\bar{w}}\,\frac{n(t)}{1+n(t)} \tag{4.19}$$

The response time, from the time instant a request arrives at the first queue to the request finishes at the processor, is modeled according to *Little's law*. The response time is measured as an average of the requests which have been completed in a sample interval. This imposes lowpass–filtering and is modeled as a continuous first–order filter,

$$\frac{d}{dt}d(t) \quad = \quad -\frac{1}{T_d}d(t) + \frac{1}{T_d}\frac{n(t)}{\lambda(t)} \tag{4.20}$$

where T_d is the filter constant, which corresponds to the sampling interval.

Models of the Measurements

It is assumed that only quantities known inside the web server are measurable.

The average number of jobs inside the web server is denoted n_i and is measurable. The dynamics are associated with the queue and not with the measurements. Because the average internal queue–length and the average queue–length of the entire queue are just different measurements of the same queue, it is assumed that the static relation from Eq. (4.13) is valid also in the dynamic case. Thus

$$n_i(t) \quad = \quad n(t)\left[1 - \left(\frac{n(t)}{1+n(t)}\right)^{M_c}\right] =: f_i\Big(n(t)\Big); \tag{4.21}$$

The arrival rate at the web server λ_i is measured simply by counting the number of arriving request over a certain time interval. The

model is not as trivial as it may seem. First, consider the external buffers, and denote the average number of jobs in the external buffers by n_e. This queue is modeled as a simple integrator, where the queue length is given by the integral of the difference of the inflow and outflow. This is similar to a conservation law from physics, as *e.g.* flow of liquid into and out from a container. The model becomes

$$n_e(t) = \int_0^t \Big(\lambda(\tau) - \lambda_i(\tau)\Big) d\tau \tag{4.22}$$

Even in the dynamic case, the entire queue–length is the sum of the internal queue–length and the external buffer–length. The static relation described in Eq. (4.6) is also valid in the dynamic case, and therefore

$$n(t) = n_i(t) + n_e(t) \tag{4.23}$$

By differentiation, the following model is obtained

$$\lambda_i(t) = \lambda(t) - \frac{d}{dt}\{n(t) - n_i(t)\} \tag{4.24}$$

Since the arrival rate is rather noisy, a linear first–order filter is applied.

$$\frac{d}{dt}\lambda_f(t) = -\frac{1}{T_\lambda}\lambda_f(t) + \frac{1}{T_\lambda}\lambda_i(t) \tag{4.25}$$

where T_λ is the filter constant.

The average response time of the web server d_i is regarded as the time from the arrival to the web server of the requests until the requests have been fully processed. Following the derivation of the model for the entire–queue response–time, the sampling is modeled as a low–pass filter. Using *Little's law*, the model becomes

$$\dot{d}_i(t) = -\frac{1}{T_d} d_i(t) + \frac{1}{T_d}\frac{n_i(t)}{\lambda_i(t)} \tag{4.26}$$

4.4 Linearization

The linearized model of the entire queue resembles the model described in the *Background* chapter (Chapter 2). This gives

$$\Delta\dot{n}(t) = -\gamma_1 \Delta n(t) + \gamma_2 \Delta p_r(t) + \Delta\lambda(t) \tag{4.27}$$

$$\Delta\dot{d}(t) = -\frac{1}{T_d}\Delta d(t) + \frac{1}{T_d}\Big(\gamma_n \Delta n(t) + \gamma_\lambda \Delta\lambda(t)\Big) \tag{4.28}$$

$$\Delta\dot{d}_i(t) = -\frac{1}{T_d}\Delta d_i(t) + \frac{1}{T_d}\Big(\sigma_n \Delta n_i(t) + \sigma_\lambda \Delta\lambda_i(t)\Big) \tag{4.29}$$

$$\Delta\lambda_i(t) = \Delta\lambda(t) + \frac{d}{dt}\{\Delta n(t) - \Delta n_i(t)\} \tag{4.30}$$

$$\Delta\dot{\lambda}_f(t) = -\frac{1}{T_\lambda}\Delta\lambda_f(t) + \frac{1}{T_\lambda}\Delta\lambda_i(t) \tag{4.31}$$

$$\Delta n_i(t) = \sigma_f \Delta n(t) \tag{4.32}$$

with the operation point given by

$$n^0 = \frac{\lambda^0 \bar{w}}{p_r^0 - \lambda^0 \bar{w}} \tag{4.33}$$

$$n_i^0 = f_i\left(n^0\right) = n^0\left[1 - \left(\frac{n^0}{1+n^0}\right)^{M_c}\right] \tag{4.34}$$

$$d^0 = \frac{\bar{w}}{p_r^0 - \lambda^0 \bar{w}} \tag{4.35}$$

$$d_i^0 = \frac{n_i^0}{\lambda^0} \tag{4.36}$$

and the parameters given by

$$\gamma_1 = \frac{p_r^0}{\bar{w}}\frac{1}{n^0+1} - \frac{p_r^0}{\bar{w}}\frac{n^0}{(n^0+1)^2} \tag{4.37}$$

$$\gamma_2 = \frac{n^0}{\bar{w}(n^0+1)} \tag{4.38}$$

$$\gamma_n = \frac{1}{\lambda^0} \tag{4.39}$$

$$\gamma_\lambda = -\frac{n^0}{(\lambda^0)^2} \tag{4.40}$$

$$\sigma_n = \frac{1}{\lambda^0} \tag{4.41}$$

$$\sigma_\lambda = -\frac{n_i^0}{(\lambda^0)^2} \tag{4.42}$$

$$\sigma_f = \frac{n^0 + 1 - n^0\left(\frac{n^0}{n^0+1}\right)^{M_c} - \left(\frac{n^0}{n^0+1}\right)^{M_c} - M_c\left(\frac{n^0}{n^0+1}\right)^{M_c}}{n^0+1} \tag{4.43}$$

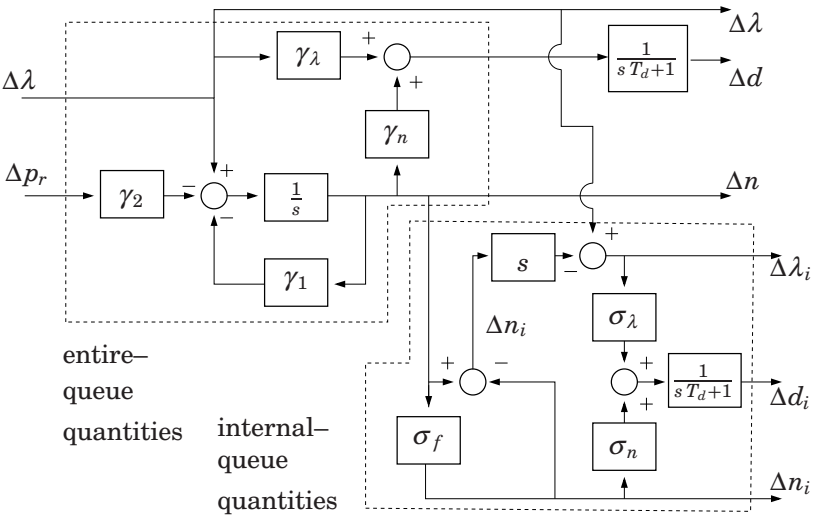

Figure 4.6 Block diagram of the server with outputs both from the entire queue and from the web server. The variables are represented in the Laplace domain.

Transfer Functions

Transfer functions reveals interesting properties of dynamic systems in the frequency domain. Transfer functions are found by transforming the linear differential functions into the Laplace domain s by the Laplace transformation, see *e.g.* [Franklin *et al.*, 1994]. The system has two inputs and a number of outputs. The work presented later in this thesis presents controllers based on measurements of the response time and the arrival rate, which are related to the inputs according to the following equations.

$$\Lambda_i(s) = \frac{\sigma_f\, s + \gamma_1}{s + \gamma_1}\,\Lambda(s) + \frac{\gamma_2(1-\sigma_f)s}{s+\gamma_1}\,P_r(s) \tag{4.44}$$

$$D(s) = \frac{\gamma_\lambda\, s + (\gamma_n + \gamma_\lambda\,\gamma_1)}{(s+\gamma_1)(s\,T_d+1)}\Lambda(s) - \frac{\gamma_n\,\gamma_2}{(s+\gamma_1)(s\,T_d+1)}\,P_r(s) \tag{4.45}$$

$$D_i(s) = \frac{\sigma_\lambda\,\sigma_f s + (\sigma_n\,\sigma_f + \sigma_\lambda\,\gamma_1)}{(s+\gamma_1)(s\,T_d+1)}\Lambda(s) - \frac{\sigma_\lambda\gamma_2(\sigma_f-1)s + \sigma_n\sigma_f\gamma_2}{(s+\gamma_1)(s\,T_d+1)}\,P_r(s) \tag{4.46}$$

Equation (4.45) reveals that the entire queuing–system can be represented by a simple second–order system from the control input p_r to the output d. Also, the disturbance λ affects the system in a fairly simple

manner. If the entire queue is not available for measurements, the control design must rely on the measurements of the internal queue. That is, the system is now described by Eqs. (4.44) and (4.46). It is observed that if the external buffers really can be neglected (σ_f=1), the measured quantities match those of the whole queue ($\lambda_i(t) = \lambda(t)$ and $d_i(t) = d(t)$). The equations also reveal that the measurement of the disturbance λ_i now becomes state dependent.

4.5 Parameter Estimation and Model Validation

The nonlinear model has been implemented in Simulink® for Matlab®. The model consists of the entire queuing–system (Eqs. (4.19) and (4.20)), the internal measurements (Eqs. (4.21), (4.24), and (4.26)). For the results presented in this chapter, the arrival–rate filter–constant in Eq. (4.25) was chosen to T_λ=10 s.

The model has only two parameters; $\bar{w}$ and M_c. The parameter $\bar{w}$ is not explicitly set in the test bed described in Chapter 3, so it has to be estimated. The parameter M_c is set explicitly by the system administrator in the Apache configuration file as the parameter `MaxClients`. Therefore, only one parameter must be estimated.

The model specifically describes the influence of the external buffers, so for the parameter estimation and for the validation it is important that the experiments conducted excite the external buffers. This is ensured by limiting the internal queue (`MaxClients` set to 80), forcing the external buffers to hold more jobs. In practice, this short internal queue is not realistic, but is chosen to demonstrate the effect of the limited internal queue–length. With a realistic value, as for example `MaxClients` set to 256, a much higher load would be required to show the same effect.

Parameter Estimation

The estimation of the required work, $\bar{w}$ is done by steady–state measurements. It is not feasible to measure the response time from the entrance of the server, but it is feasible to measure the response time at the clients in steady–state. This measurement includes the network delay, which is not part of the model. It is assumed that the response time experienced at the clients is the average round–trip–time added to the response time of the web server. The average round–trip–time is used as offset to the response times of the entire queue, predicted by the model.

To excite the external buffers, the web server was operating close to the stability limit (in the queuing–theoretical sense). Longer experiments with CPU capacity of around 26% of the total capacity and arrival rate of

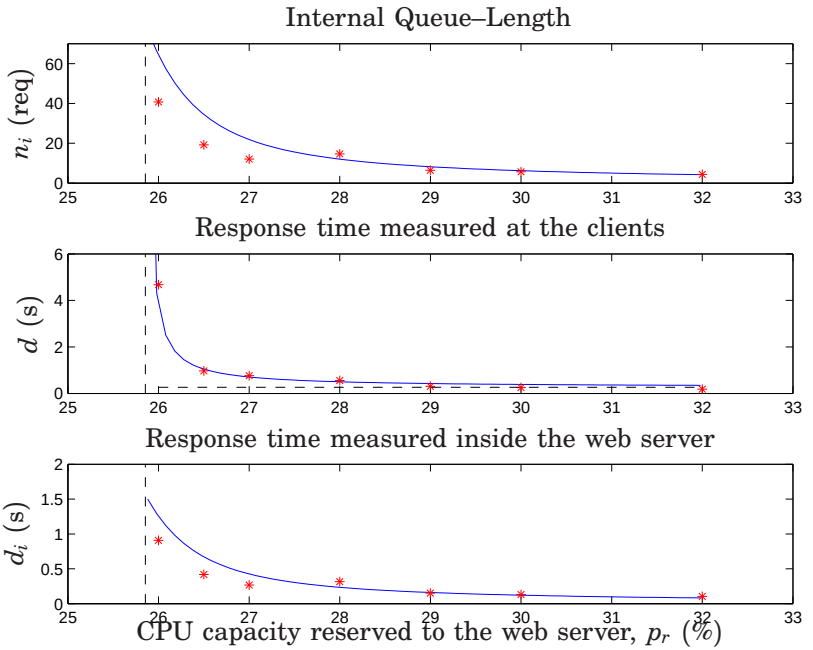

Figure 4.7 Validation of the model versus experimental data. The full line represents the model, and the stars represent steady–state experiment data of approximately 2000 s (2000 measurements). The vertical dashed lines represent the theoretical limit for queuing stability (where $\lambda=\mu$). The horizontal dashed line in the middle figure represents the estimated network round–trip–time.

around 50 req/s showed a stable, but highly loaded system, whereas CPU capacity of around 25% of the total capacity and arrival rate of around 50 req/s resulted in an overloaded system.

Several steady–state experiments with arrival rate of 51.3 req/s were conducted with different CPU capacities. Transients were removed, leaving approximately 2000 measurements for averaging. The sampling interval was 1 s.

Figure 4.7 shows the results of the experiments along with the theoretic values obtained with the value $\hat{w}$=0.00504 s, which was considered as the best model fit. The steady–state model is sensitive to the choice of $\hat{w}$ at loads close to the stability limit, but this is a known problem in the field of queuing systems. The parameter also enters the dynamic properties of the system, but here the sensitivity is not as profound. The objective of the parameter estimate was to match the model to the experimental data for the entire queue as this model is well established within

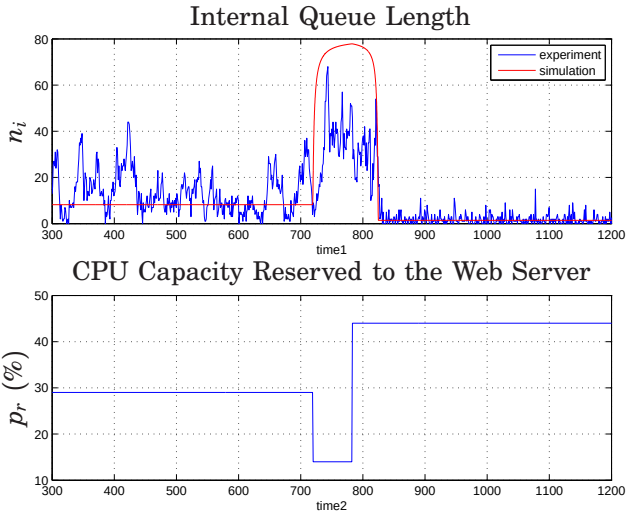

Figure 4.8 Validation of the simulated model versus experimental data. The lower part of the figure shows the control signal p_r applied for both the simulation and the experiment. The top of the figure shows the behavior of the internal queue.

the queuing–theory community (see the middle of Fig. 4.7). The lower part of the same figure shows the response times measured inside the web server. The match between the model and the measurements are not as accurate as in the top of the figure, but it is still acceptable. The queue length of the entire queue was not measurable, so comparison is only conducted for the internal queue illustrated in the top of Fig. 4.7. Also here, some deviation is observed, but the model follows the experimental results quite well.

Model Validation

The dynamic model was validated towards experimental data. To show the dynamic influence of the external buffers, a scenario was conducted as both experiment and simulation. The arrival rate was kept constant at 51.3 req/s throughout the entire experiment, while the control signal p_r was as illustrated in the top of Fig. 4.8. The results of the experiment are illustrated in Figs. 4.8 and 4.9 along with the corresponding simulation results. The top of Fig. 4.8 shows the behavior of the internal queue. When the control signal was decreased and the server was overloaded, the queue built up, but because it was limited to 80 requests, it did not grow that much. The top of Fig. 4.9 shows the effect of the queue building up; the response times became longer, but still, the full effect was not seen, since

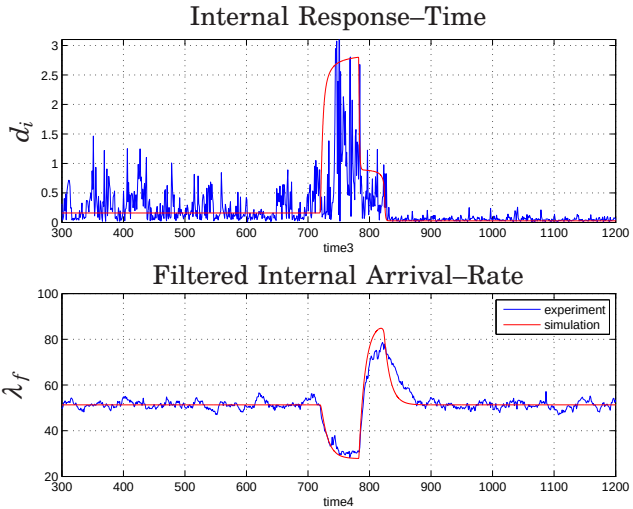

Figure 4.9 Validation of the simulated model versus experimental data. The upper part of the figure illustrates the response time measured inside the web server. The lower part of the figure shows the filtered arrival rate measured inside the web server. (The filter is a linear first–order filter with a time constant of 10 s).

the response times of the entire queue could not be measured. The bottom of Fig. 4.9 reveals some interesting effects of the external buffers. When the control signal was decreased, the service rate became smaller than the arrival rate. Since the internal buffer was often full, the rate of jobs entering the web server was limited by the service rate, and therefore the measured internal arrival–rate decreased. When the control signal was increased, the flow into the web server was still limited by the service rate until the external buffer was emptied. At this time, the service rate was around 87 req/s, which is clearly observed as the top peak in the bottom of Fig. 4.9. When the queue had settled at the new operating point, the arrival rate measured inside the web server matched that of the offered. It is worth to remember that the arrival rate entering the entire queuing system was constant during the whole experiment.

The match between the simulated model and the measurements is acceptable. The problems of matching the correct levels of response times and internal queue length observed in the steady–state experiments remains, but the dynamics of the system are matched quite well by the model. In particular, the filtered internal arrival–rate can be matched well.

4.6 Discussions and Conclusions

This chapter has considered the problem where several buffers, denoted the external buffers, were inserted between the client and the web server, and the variables of the external buffers were unmeasurable. The chapter investigated the dynamic effects of these extra buffers, when measurements were taken inside the web server, and the investigations have revealed an interesting property:

- The (internal) measurement of the arrival rate becomes state dependent.

This alters the properties of a controlled system significantly, so neglecting the external buffers in the control design can become devastating. If the measurement of the arrival rate is assumed to be independent of the server system, a feed–forward mechanism is easily designed to improve transient performance. As this measurement turns out to be state dependent, the feed–forward mechanism changes to a feedback mechanism, which can compromise the stability of the system.

The model has only two parameters; one explicitly set by the system administrator (M_c) and one which has to be estimated ($\bar{w}$). The model has been verified by comparing simulated behavior to experimental data and good correspondence has been shown. The levels of some of the variables were sensitive to the estimated parameter when the system was operated close to overload. This is a normal situation within queuing systems. The dynamic properties of the model was not as sensitive to the parameter variation.

5

Control Design and Analysis

The work presented in this chapter is based on the modeling results of external buffering for web servers presented in the previous chapter. To stress the importance of the suggested model, controllers are designed with the assumption that the external buffers can be neglected, which if that was true, would mean that a feed–forward control could be used to reduce the effect of a disturbance without compromising the stability. However, analysis shows that the stability is indeed compromised when the effect of the external buffering is taken into account. This problem is relevant as a modern computer system contains many buffers in the underlying structure, which are often not well known by the control designer. The material presented in this chapter is based on [Kjær and Robertsson, 2009; Kjær and Robertsson, 2010].

If the web–server system is operated in medium or light load, the external buffers can be neglected simplifying the measurements and the control design significantly. The question is then how the controlled system behaves when operating under higher loads where the external buffers become significant.

First, a control strategy based on both feedback from the output d and feed–forward from the disturbance λ is considered, where it is assumed that the external buffers can be neglected. The response time is chosen as main metric since it is important from a client–perspective. Another

choice could be queue–length control, which is more oriented towards the operator. An analysis then follows, where the control structure and the control parameters remains the same but the system now includes the external buffers.

5.1 Control Design Neglecting the External Buffers

The system is modeled by a second–order system with two inputs; one input for control action p_r, and one input as a disturbance λ. The linear model was derived in Chapter 4. Equation (4.45) is repeated here for convenience

$$D(s) = \frac{\gamma_\lambda\, s + (\gamma_n + \gamma_\lambda\, \gamma_1)}{(s+\gamma_1)(s\,T_d+1)}\Lambda(s) - \frac{\gamma_n\,\gamma_2}{(s+\gamma_1)(s\,T_d+1)}\,P_r(s) \quad (5.1)$$

The low–pass filter $\left(1/(s\,T_d{+}1)\right)$ represents the averaging of the response–time during a sample interval. The parameter T_d is the size of the sampling intervals.

Feedback Design

Applying a PI controller as feedback ensures that the reference d_r is reached in steady state. The PI controller is defined as [Åström and Hägglund, 2005]

$$P_{fb}(s) = -\frac{K\ (T_i s+1)}{T_i s}\left(D_r(s) - D(s)\right) \quad (5.2)$$

The first minus–sign is included to compensate for the negative gain from the control signal to the output of the server model. Applying this controller, the closed–loop expression for $D(s)$ becomes

$$\begin{aligned} D(s) = {} & \frac{\gamma_2\,(\gamma_n\,K\,T_i\,s + \gamma_n\,K)}{T_iT_ds^3 + T_is^2 + T_iT_d\gamma_1s^2 + T_i\gamma_1s + \gamma_n\gamma_2KT_is + \gamma_n\gamma_2K}D_r(s) \\ & + \frac{T_i\,(\gamma_\lambda\,s + \gamma_\lambda\,\gamma_1 + \gamma_n)\,s}{T_iT_ds^3 + T_is^2 + T_iT_d\gamma_1s^2 + T_i\gamma_1s + \gamma_n\gamma_2KT_is + \gamma_n\gamma_2K}\Lambda(s) \end{aligned} \quad (5.3)$$

The essential properties of a PI–controlled system are also seen here. A step–change in the reference d_r is followed by the response time d in steady–state, and a disturbance λ will be removed in steady–state, assuming that the closed–loop system is stable.

Feed–Forward Design

The disturbance λ is measured and utilized for feed–forward control to reduce the undesired effects of changes in the arrival process. The term *feed–forward* is used to emphasize that the only measurement used for this controller is an input (the disturbance)—no outputs are used. Later, the measurement used for the feed–forward controller will be exchanged with an internal measurement of the arrival rate, which can be considered as an output of the process. The feed–forward mechanism will then actually become a feedback mechanism but it will still be denoted as *feed–forward* to state that it was originally intended as a feed–forward mechanism.

Because the measurement of the disturbance can be quite noisy, it is filtered with the first–order low–pass filter from Eq. (4.25).

The feed–forward controller is based on the queuing–theoretical relationships from Eqs. (4.15) and (4.17). The equations are rewritten to a linear function of the filtered arrival rate λ_f and a constant term given by the desired response time d_r and the estimated required work. The estimated required work, $\bar{w}$ is treated as a constant, which must be estimated, and is denoted $\hat{w}$.

$$p_{ff}(t) = \frac{\hat{w}}{d_r} + \hat{w}\,\lambda_f(t) \tag{5.4}$$

where p_{ff} is the control signal derived from the feed–forward controller. This feed–forward mechanism is a proportional controller with proportional gain $\hat{w}$ and a bias term $\hat{w}/d_r$. It is noted that the bias–term is proportional to $\hat{w}$, which makes it difficult to evaluate the performance with different values of $\hat{w}$ since the operation point changes with $\hat{w}$. Therefore, when applying the feed–forward controller only (without the feedback controller), the feed–forward expression is rewritten into

$$p_{ff}(t) = p_r^0 + \hat{w}\left(\lambda_f(t) - \lambda^0\right) \tag{5.5}$$

In the case where the PI controller is also active, the integral part will handle the deviation in the bias term.

Linearization and Laplace transformation of the feed–forward controller gives

$$P_{ff}(s) = \frac{\hat{w}}{T_\lambda s + 1}\Lambda(s) \tag{5.6}$$

The control system consisting of both feed–forward and feedback con-

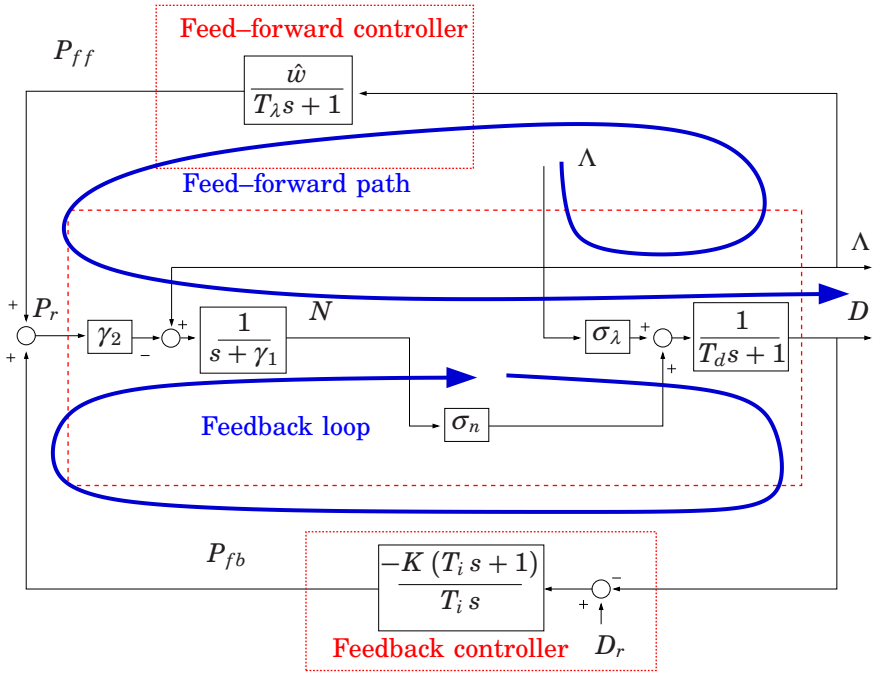

Figure 5.1 Block diagram of the server with feedback and feed–forward control from the real disturbance.

trollers is illustrated in Fig. 5.1 and has the form

$$D(s) = \frac{\gamma_2\,(\gamma_n\,K\,T_i\,s + \gamma_n\,K)}{T_i T_d s^3 + (T_i + T_i T_d \gamma_1)s^2 + (T_i\gamma_1 + \gamma_n\gamma_2 K T_i)s + \gamma_n\gamma_2 K} D_r(s) + \frac{T_i\,(\gamma_\lambda\,T_\lambda\,s^2 + (\gamma_\lambda + \gamma_\lambda\,\gamma_1\,T_\lambda + \gamma_n\,T_\lambda)\,s + (\gamma_\lambda\,\gamma_1 - \gamma_n\,\gamma_2\,\hat{w} + \gamma_n))\,s}{(T_\lambda s+1)(T_i T_d s^3 + (T_i + T_i T_d \gamma_1)s^2 + (T_i\gamma_1 + \gamma_n\gamma_2 K T_i)s + \gamma_n\gamma_2 K)} \Lambda(s) \tag{5.7}$$

which holds the same properties as described for the feedback controller, but with the possibility to alter the transient response of the disturbance rejection by changing $\hat{w}$ and T_λ.

5.2 Stability Analysis Including the External Buffers

Consider the case where the control design is conducted as described above (assuming that the external buffers can be neglected), but now the exter-

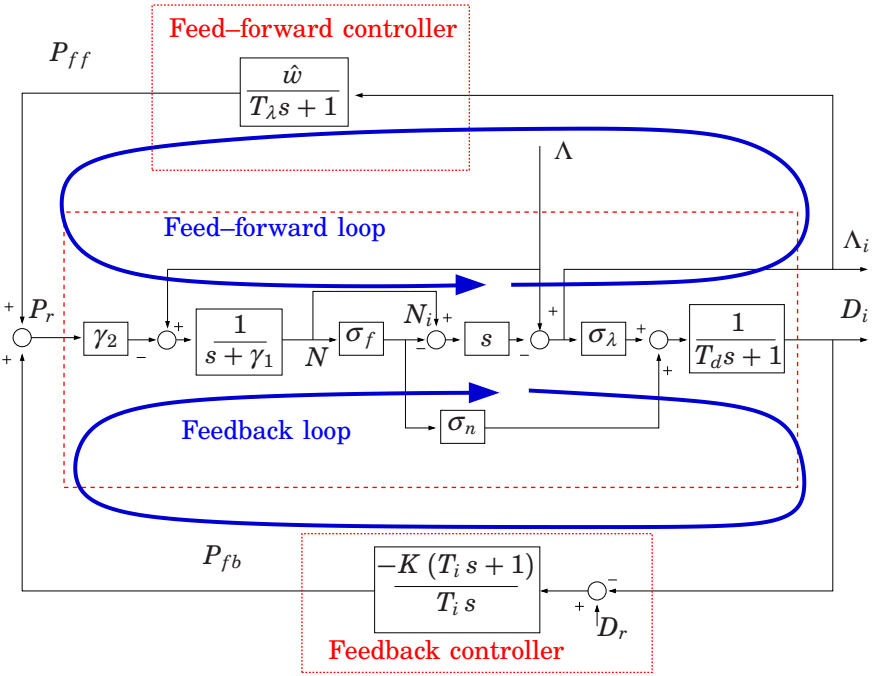

Figure 5.2 Block diagram of the server with feedback and feed–forward from the state dependent measurement of the internal arrival–rate.

nal buffers are significant. The analysis then follows that of the above except that λ and d are exchanged with λ_i and d_i. This has the consequence that an extra zero is included in the server model, and the arrival rate λ , which before was treated as a disturbance, now becomes a dynamic state (λ_i), which introduces an extra feedback loop as illustrated in Fig. 5.2.

The expressions for stability analysis becomes too general to draw any conclusions from if no parameters are fixed. Therefore, it is in this thesis chosen to evaluate the stability for the testbed described in Chapter 3, and for which the parameters are estimated in Chapter 4. The parameters and operating point are chosen according to the following arguments.

The model parameters are chosen according to the system identified in Section 4.5; $M_c = 80$ and $\hat{w} = 0.00504$ s.

The operation point is chosen as an acceptable response time for the end user. Because the entire response–time cannot be measured, a reference is defined for the internal response–time. It is chosen as

Table 5.2 Operating points

Parameter	p_r^0	λ^0	μ^0	n^0	n_i^0	d^0	d_i^0
Unit	%	req/s	req/s	req	req	s	s
Value	26.16	51.30	51.91	83.56	51.3	1.63	1.0

$d_i^0 = 1.0$ s. Also, a constant arrival rate of $\lambda^0 = 51.3$ rad/s is chosen, as this was the arrival rate used for the parameter estimation for the model. The remaining operating–point parameters are listed in Table 5.2.

The feedback–control parameters are chosen to $T_i = 100.0$ and $K = 0.1$. These values are chosen to give large robustness margins (infinite amplitude margin and a phase margin of almost 90°) when neglecting the external buffers. When taking the zero induced by the external buffers into account, the phase margin is reduced to 71° while the amplitude margin becomes 10.8 dB, which is still on the conservative side. The cross–over frequency, which is a metric for the closed–loop dynamics, is 0.38 rad/s and 0.1 rad/s neglecting and including the external buffers, respectively. Because the arrival rate can be quite irregular, a relatively high degree of filtering is imposed. A value of $T_\lambda = 10$ s is considered to give a nice behavior of λ_f.

The chosen operating point represents a highly loaded situation, which matches the objective to maximize the utilization of the CPU.

Analysis of the Feedback Design

Due to the complicated denominator, it is hard to say anything general about how the different parameters affect the stability.

Eqs. (4.45) and (4.46) represent the server system from the control signal to the output for the system with unlimited and limited measurements, respectively. Fig. 5.3 illustrates the Bode diagrams for the two systems when the PI controller is applied. The figure indicates that the limited measurement does not affect the dynamic properties significantly for low and middle frequency range. For higher frequencies, the phase curves deviate, but here the gain is so low that the phase is no longer relevant. Therefore, the stability and the dynamic properties of the system does not really change when the response time is measured inside the server instead of outside the server.

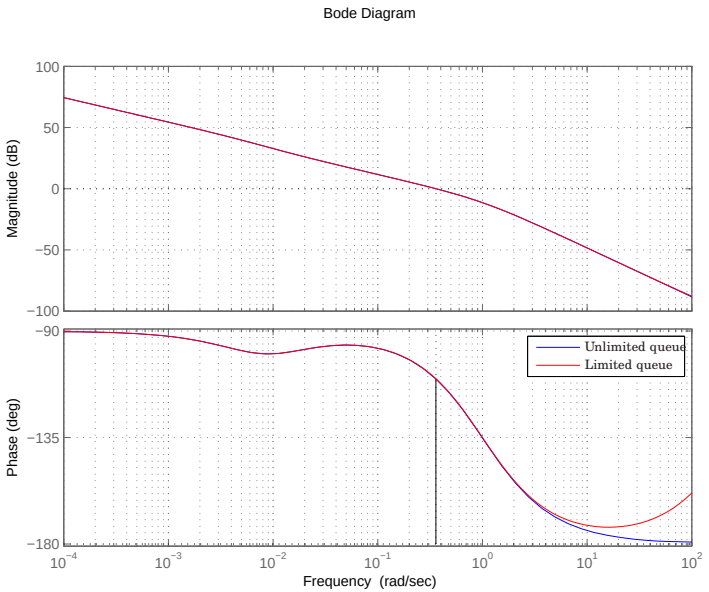

Figure 5.3 Bode diagrams for the PI–controlled server. The blue lines represent the system without limited measurements (Eq. (4.45)) and the red line illustrates the system with limited measurements (Eq. (4.46)). The black line indicates the phase margin which is 70°.

Analysis of the Feed–Forward Design

Using the feed–forward controller alone introduces a feedback from the number of jobs in the system, which alters the stability properties of the system. The system is then described by

$$\frac{D_i(s)}{\Lambda(s)} = \frac{\sigma_\lambda\,\sigma_f\,T_\lambda\,s^2 + (\sigma_\lambda\,\gamma_1\,T_\lambda + \sigma_\lambda\,\sigma_f + \sigma_n\,\sigma_f\,T_\lambda)\,s + \sigma_\lambda\,\gamma_1 + \sigma_n\,\sigma_f - \sigma_n\,\sigma_f\,\gamma_2\,\hat{w}}{(sT_d + 1)\,(T_\lambda\,s^2 + (1 - \gamma_2\,\hat{w}(1 - \sigma_f) + \gamma_1\,T_\lambda)\,s + \gamma_1)} \tag{5.8}$$

which indicates that the system becomes unstable for

$$\hat{w} > \frac{\gamma_1\,T_\lambda + 1}{\gamma_2\,(1 - \sigma_f)} \tag{5.9}$$

The test–bed system is, according to the parameters chosen in the begin-

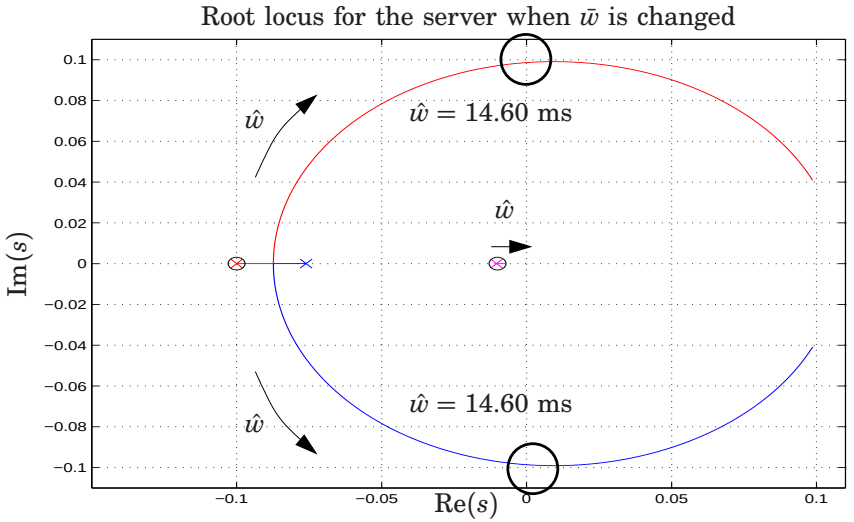

Figure 5.4 Root locus for the server when $\hat{w}$ used in the feed–forward controller is changed from 0 ms to 30 ms. A fixed PI–controller is also imposed. A stable pole is also located in $s = -1.20$ which slowly moves towards the origin..

ning of this chapter, unstable for

$$\hat{w} > 7.281 \text{ ms} \tag{5.10}$$

Analysis of the Combined Feedback and Feed–Forward Design

The complete transfer–function for the system containing both the feedback and the feed–forward mechanism is too comprehensive to present on paper, so only the transfer–function representing the test bed for $\hat{w} = 5.04$ ms is presented here.

$$\frac{D_i(s)}{\Lambda_i(s)} = \frac{0.2834s^2 - 0.05678s + 0.0003822}{s^4 + 1.32s^3 + 0.1604s^2 + 0.01147s + 0.000095} \tag{5.11}$$

Figure 5.4 shows the root–locus when all parameter but $\hat{w}$ in the feed–forward part are kept constant. The analysis suggests that this system is unstable for

$$\hat{w} > 14.06 \text{ ms} \tag{5.12}$$

The stability limit has been increased by including the feedback loop but only to a certain degree. This is not a general result but it is only valid for the system resembling the test bed.

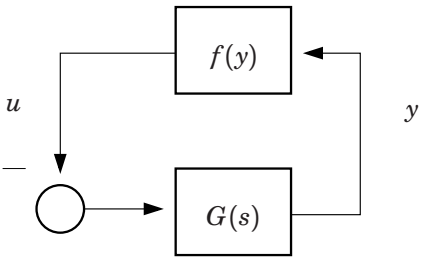

Figure 5.5 Block diagram of an interconnection between a static nonlinearity and a linear dynamic system. The describing–function method requires this form of interconnection.

5.3 Classification of Instability

In systems with limitations (which are always present in practical applications) instability can expose itself in two different ways; either the system goes to an extreme configuration determined by the limitations of the system (such as saturated actuator, saturated states, etc.) and remains there until external events (such as user interaction) occur, or the system starts to oscillate. Methods to predict this type of instability exist and one such method is the describing–function method, which will be described briefly below.

The Describing–Function Method

Consider an interconnected system as in Fig. 5.5 where $G(s)$ is a linear system and $f(y)$ is a static nonlinearity. Normal linearization of the nonlinearity only reveals whether the system is locally asymptotically stable or not—not necessarily the character of the possible instability. The describing–function method continues the idea of linearization but instead of linearizing around an equilibrium, it is assumed that the system will converge towards a limit cycle. The describing function is a description of how the nonlinearity reacts to an oscillating input. The nonlinearity with a sinusoidal input is approximated by a Fourier series and the higher–order term are neglected. The describing function can then be combined with the linear system in the frequency–domain to predict whether there will be a stable or unstable limit cycle at a given amplitude and frequency. The formulation presented here follows the notation used by [Slotine and Li, 1991] but also [Khalil, 2002] gives a comprehensive treatment.

The describing function $\Psi(A)$ of a static nonlinearity $f(\cdot)$ is computed

by

$$a_1 = \frac{1}{\pi}\int_{-\pi}^{\pi} f\Big(A\,\sin(\omega t)\Big)\cos(\omega t)\,d(\omega t) \tag{5.13}$$

$$b_1 = \frac{1}{\pi}\int_{-\pi}^{\pi} f\Big(A\,\sin(\omega t)\Big)\sin(\omega t)\,d(\omega t) \tag{5.14}$$

$$\Psi = \frac{b_1 + a_1 i}{A} \tag{5.15}$$

Computation of the Describing Function for the Web Server

Several nonlinearities are present in the server system. The nonlinear model prevents the queue from holding negative jobs ($n < 0$) but this is not captured by the linear model. The second nonlinearity is an actuator saturation since CPU capacity is limited from both below and from above. The limitations may be defined by only the physical limitations of the CPU, but further limitations may be imposed by the system design (such as reserving a certain amount of CPU capacity for administrative tasks). This nonlinearity is not relevant for the investigated operating–point, as the control signal remains within the available range. Therefore, the actuator nonlinearity is not treated any further. The third nonlinearity is the static function of Eq. (4.21), relating the internal queue–length to the full queue–length, which holds for $n \geq 0$. In the case where $n < 0$, the function must be limited from below, so that $n_i \geq 0$. Furthermore, the function is altered to operate on the linearized variables:

$$\tilde{f}_i(\Delta n) := \begin{cases} (\Delta n + n^0)\left[1 - \left(\dfrac{\Delta n + n^0}{1 + \Delta n + n^0}\right)^{M_c}\right] - n_i^0 & \text{if } \Delta n \geq -n^0 \\ -n_i^0 & \text{otherwise} \end{cases} \tag{5.16}$$

This static nonlinearity is illustrated in Fig. 5.6 for different values of M_c. Assuming the same operation point ($n^0 = 70$ req in this case) the nonlinearity becomes more and more dominating as M_c is decreased. For high values of M_c the nonlinearity becomes almost linear around the operating point.

The describing function is not derived analytically here. Instead, it is computed numerically and the results are shown in Fig. 5.7 for different values of M_c.

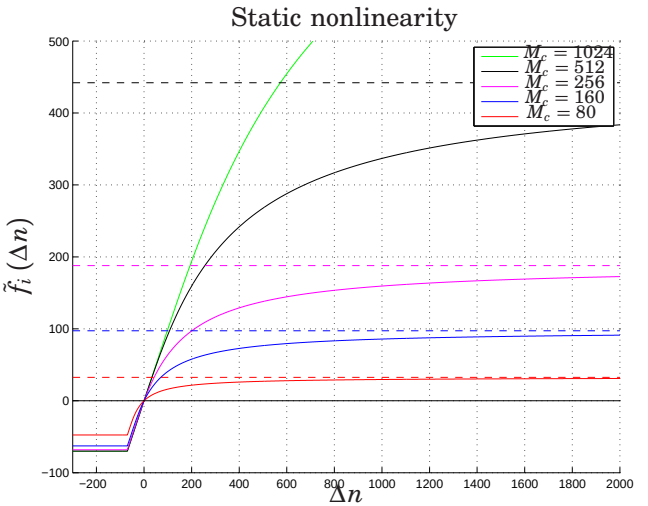

Figure 5.6 Static nonlinearity in the linearized coordinates with $n^0 = 70$, for different values of M_c. The dashed lines indicates the maximal values of the function for the different values of M_c.

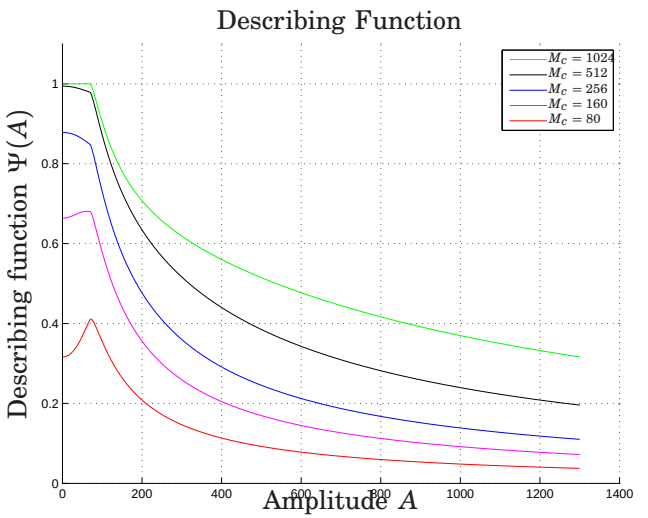

Figure 5.7 Describing function for $n^0 = 70$ for different values of M_c.

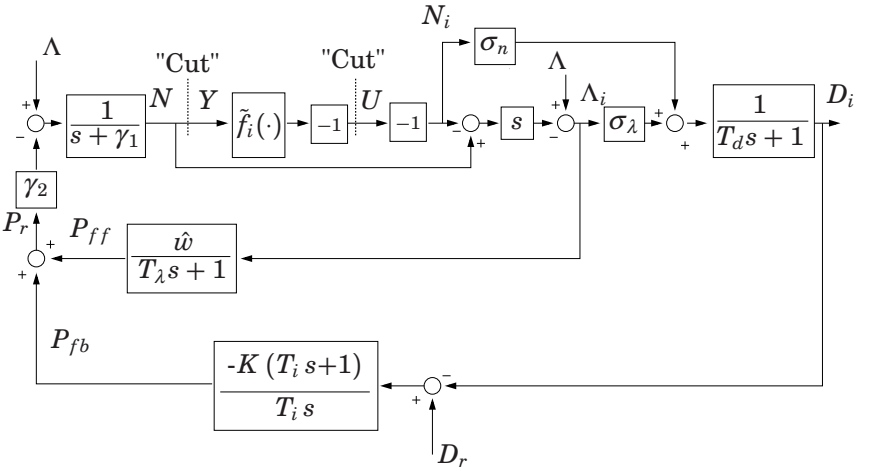

Figure 5.8 Block diagram of the server with feedback and feed–forward control.

Applying the Describing–Function Method

The closed–loop system, including both feedback and feed–forward mechanisms, illustrated in Fig. 5.2 is exchanged with the one in Fig. 5.8, where the static nonlinearity now is present instead of its linearized function. “Cuts" are made on each side of the nonlinearity. The two variables $U(s)$ and $Y(s)$ are inserted at the places of “cutting” and the whole block diagram is rearranged, so these variables become input and output of a linear system $G(s)$ as illustrated in Fig. 5.9.

Feed–Forward Control

When the feedback is not utilized, the transfer function reduces significantly (as K=0). The transfer function from $U(s)$ to $Y(s)$ is now

$$G(s, K=0) = \frac{\gamma_2\,\hat{w}\,s}{T_\lambda\,s^2 + (1-\gamma_2\,\hat{w} + T_\lambda\,\gamma_1)s + \gamma_1} \tag{5.17}$$

which has a zero in the origin and two poles, which implies that the phase of $G(s)$ will be in the interval $]-90°\ 90°[$. As the negative inverse describing–function is real and strictly negative, it will never intersect with the the Nyquist–curve and thus, the describing–function method can not predict any limit–cycle. The conclusion is therefore that since the system is unstable (for $\hat{w} > 7.281$) and without limit cycles, the system will go towards infinity (or minus infinity) until some other saturation is reached.

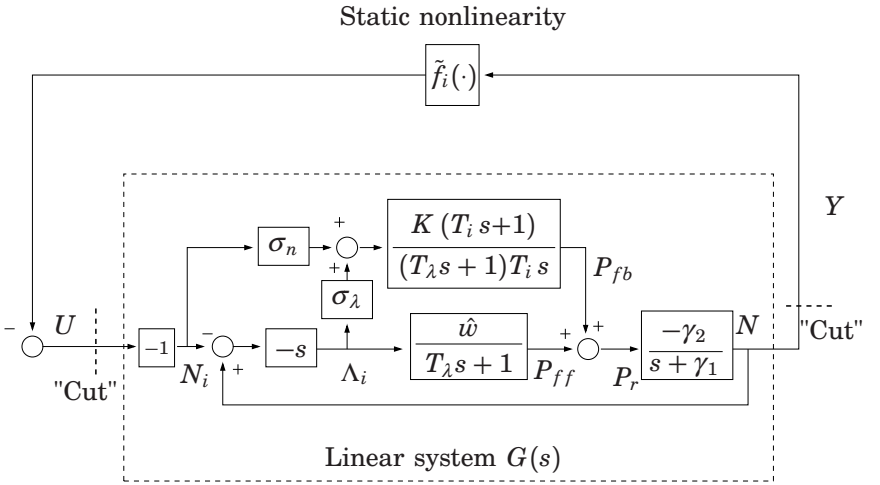

Figure 5.9 Rewritten block diagram in Fig. 5.8. The system is divided into a linear part and a nonlinear part interacting through the signals $U(s)$ and $Y(s)$.

Feedback and Feed–Forward Control

For $\hat{w} = 15$ ms the transfer function becomes

$$G(s) = \frac{-0.0881s^3 + 0.6343s^2 + 0.04166s + 0.0003822}{s^4 + 1.195s^3 - 0.1441s^2 + 0.001108s} \tag{5.18}$$

which is an unstable system with two unstable poles, one stable pole, and one pole in the origin. The right–hand side of the Laplace–plane is encircled by a closed curve S as illustrated in Fig. 5.10. The origin is not included in the encapsulation as is the normal procedure when evaluating Nyquist–diagrams. The large circle–fraction has a sufficiently large radius to be approximated as infinite (compared to the system dynamics). Note that the system has two unstable poles encapsulated by the closed curve.

The transfer–function $G(s)$ is evaluated along the curve S, and mapped into a new complex–plane to draw the Nyquist diagram, which is illustrated in Fig. 5.11.

A detailed magnification of the region–of–interest is illustrated in Fig. 5.13 along with the negative inverse describing–function. As the amplitude A increases, the negative inverse describing–function first moves towards the origin, but from around $-1/\Psi = -3.3$ it moves towards $-\infty$. This is illustrated by the blue arrow in the figure. Figs. 5.12 and 5.13 show that the two curves intersect twice at $A = 27.4$ and $A = 152.4$ ($G(\omega i)$=-1/$\Psi(A)$=2.868). The frequency of the transfer function at the intersection is $\omega = 0.101$ rad/s. Each of the intersections is a possible

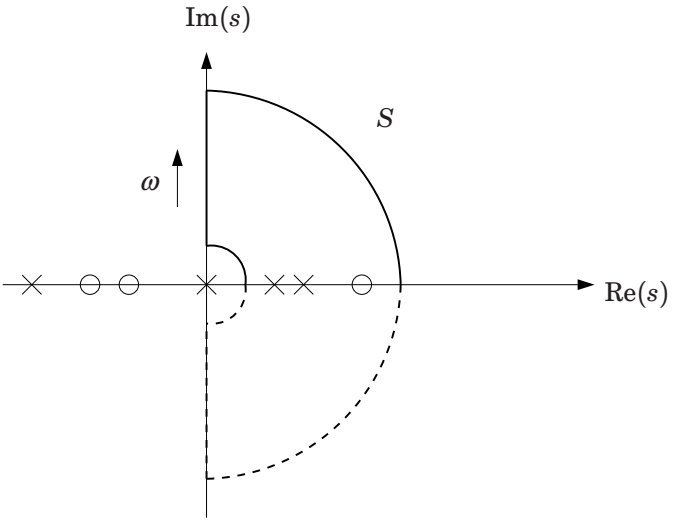

Figure 5.10 The Laplace–plane. The Nyquist curve is found by evaluating the transfer function along the curve S encapsulating the right–half plane. Note that the origin is not included.

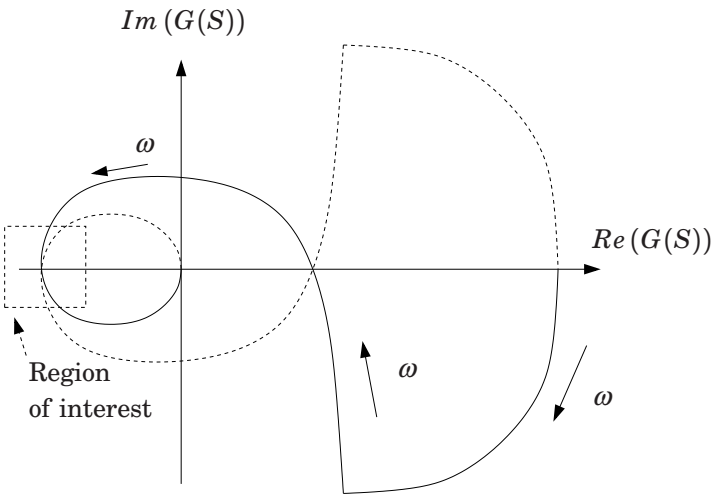

Figure 5.11 The Laplace–plane. The Nyquist curve when both the feed–forward and the feedback controller are included. The dashed line represents the mapping of the dashed part of S in Fig. 5.10.

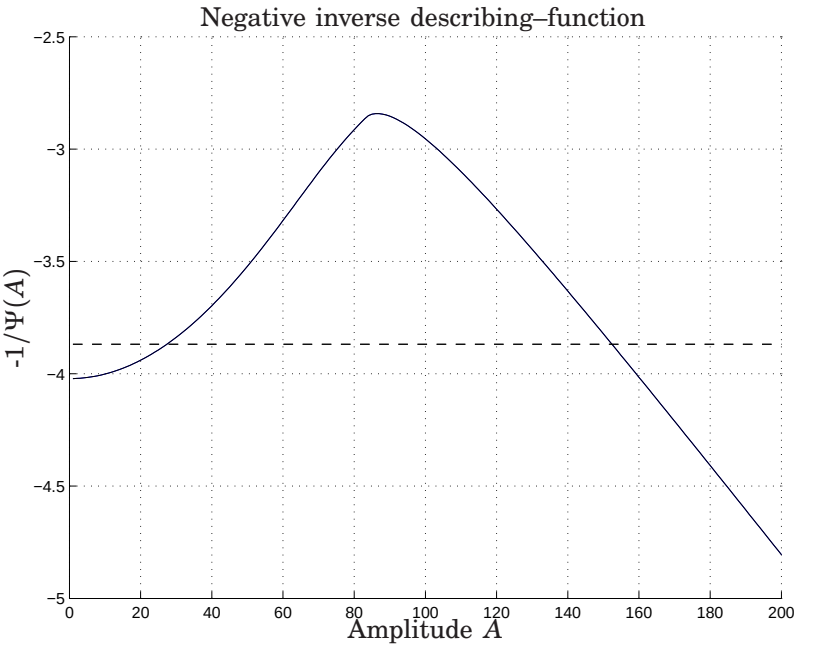

Figure 5.12 Negative inverse describing–function for M_c=80. The dashed line represents the intersection with the Nyquist–diagram ($G(0.101i) = -3.868$).

limit cycle, which is now investigated separately:

***A*=27.4:** Assume that the system is oscillating with an amplitude slightly above 27.4. The negative inverse describing–function will then have moved slightly towards the origin, and this point is encircled clockwise twice by the Nyquist curve. Following the procedure of [Slotine and Li, 1991], the limit cycle is stable because the open–loop system $G(s)$ has two poles in the right–hand side of the Laplace plane.

***A*=152.4:** Assume that the system is oscillating with an amplitude slightly above 152.4. The negative inverse describing–function will then have moved slightly towards $-\infty$, and this point is not encircled by the Nyquist curve. Following the procedure of [Slotine and Li, 1991], the limit cycle is thus not stable because the open–loop system $G(s)$ has two poles in the right–hand side of the Laplace plane.

From the arguments above, the system is expected to have a stable limit

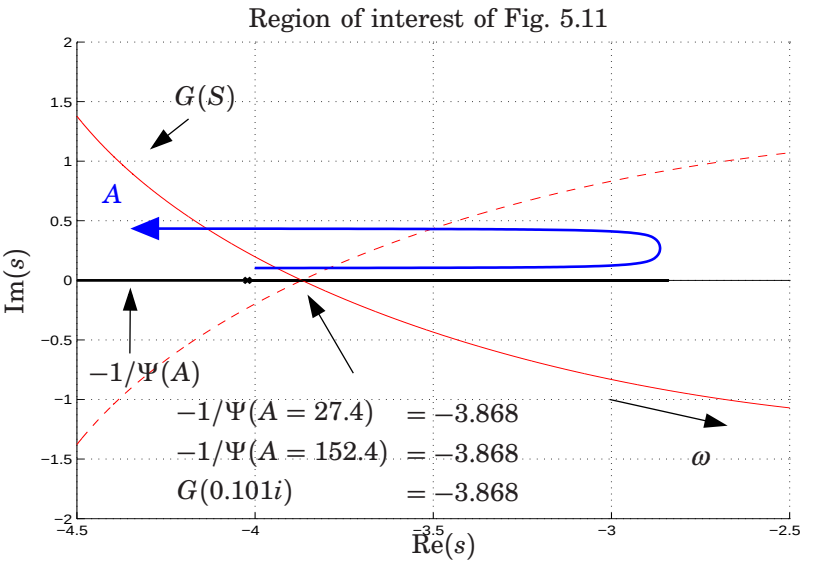

Figure 5.13 Close–up of the region of interest of Fig. 5.11 for $\hat{w} = 15$ ms along with a plot of the negative inverse describing–function $(-1/\Psi(A))$.

cycle with oscillations in the entire queue length n of 27.4 req in amplitude and with period time of approximately $2\pi/0.101 = 62.21$ s. This means that the number of jobs in the system will oscillate and not converge to a steady value, even if all inputs are constant.

Summary for the Analysis

The analysis suggests that if the value of the estimated required work, $\hat{w}$, is chosen too high, the systems will become unstable. In the saturation where only feed–forward is applied, the number of requests in the server will go towards either zero or the maximum value (not yet determined), when $\hat{w}$ becomes above 7.28 ms. When both feedback and feed–forward control is applied, the instability manifests itself as oscillations when the stability limit is reached at $\hat{w} = 14.06$ ms.

5.4 Verification by Simulation

The simulation–model developed for validation, described in Chapter 4, was expanded to include both the feed–forward and the feedback controller. A saturation was imposed to limit the control signal p_r between one and zero. The focus here is the stability and therefore all simulations were initiated at the operation point, but a small deviation in the control signal in the beginning of the simulation ensured that instability revealed itself. The operating point was as described in the beginning of this chapter (listed in Table 5.2), and the system was simulated with two control–configurations; with feed–forward control only and with the combined feedback and feed–forward controller.

Feed–Forward Control

Figure 5.14 illustrates the results of two simulations of the system with the feed–forward controller only. The figure indicates that the system remained stable and well behaved for low values of $\hat{w}$ but became unstable for higher values. The stability analysis suggested that the limit for stability is 7.281 ms which corresponds well with the presented simulation–results. Also, the simulation results indicate that the system does not enter stable limit cycles in the unstable operating point, but goes to an extreme saturation, where the control signal goes to the lower limitation and the response times goes to infinite (as the queue grows to infinity). This matches the conclusion from the describing–function analysis in Section 5.3.

Feedback and Feed–Forward Control

Fig. 5.15 illustrates the results of two simulations, where both the feedback and the feed–forward controllers were utilized. The figures indicate that the system remained stable and well behaved for low values of $\hat{w}$, but entered stable limit cycles for higher values. The limit cycles were characterized by period times of approximately 65 s. All variables oscillated, but not symmetrically around the operation–point value. Table 5.3 lists the deviations from the operation points for the most important variables. The stability analysis suggested that the limit for stability is 14.60 ms, which corresponds well with the presented simulation results. The predicted limit–cycle period was 62.21 s, which corresponds well with the period–time obtained by the simulations. The analysis also suggested that the total number of requests would oscillate with an amplitude of 27 req. The simulations suggested that the amplitude was in the range of 35 req, which shows that the simulation corresponds reasonable well with the analysis.

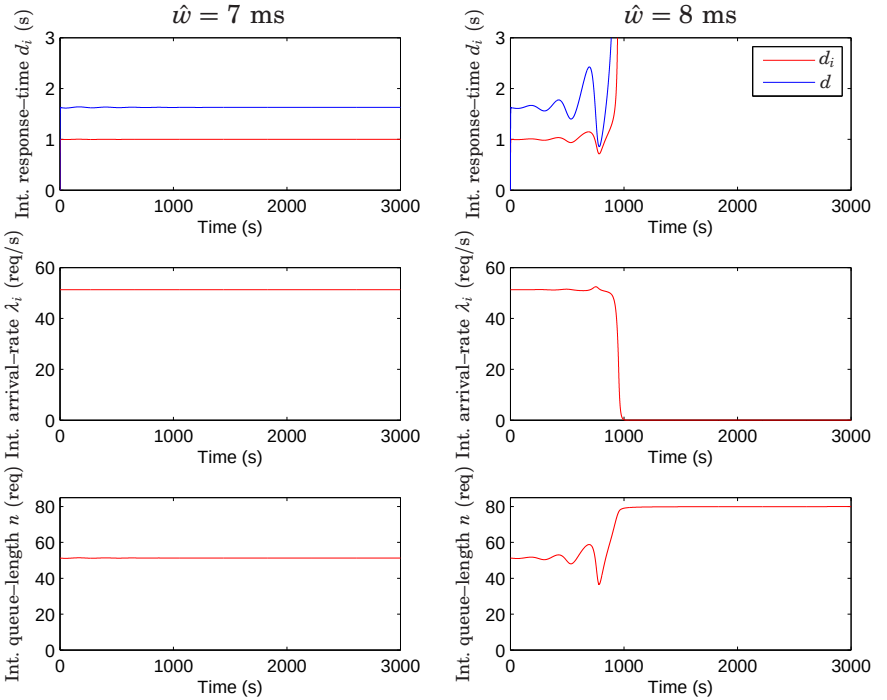

Figure 5.14 Simulation results showing the internal measured response–time d_i, the internally measured arrival rate, and the internally measured queue length when utilizing only feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller was 7 ms in the simulation results presented on the left–hand side, and 8 ms in the simulation results presented on the right–hand side.

Table 5.3 Deviation from the operating point for the oscillating system when both feed–forward and feedback is applied.

Parameter	λ_f	d_i	n	n_i
Operation point	51.3 req/s	1.0 s	83.56 req	51.3 req
Upper deviation	2.22 req/s	0.145 s	38 req	7.2 req
Lower deviation	1.59 req/s	0.228 s	34 req	11.7 req

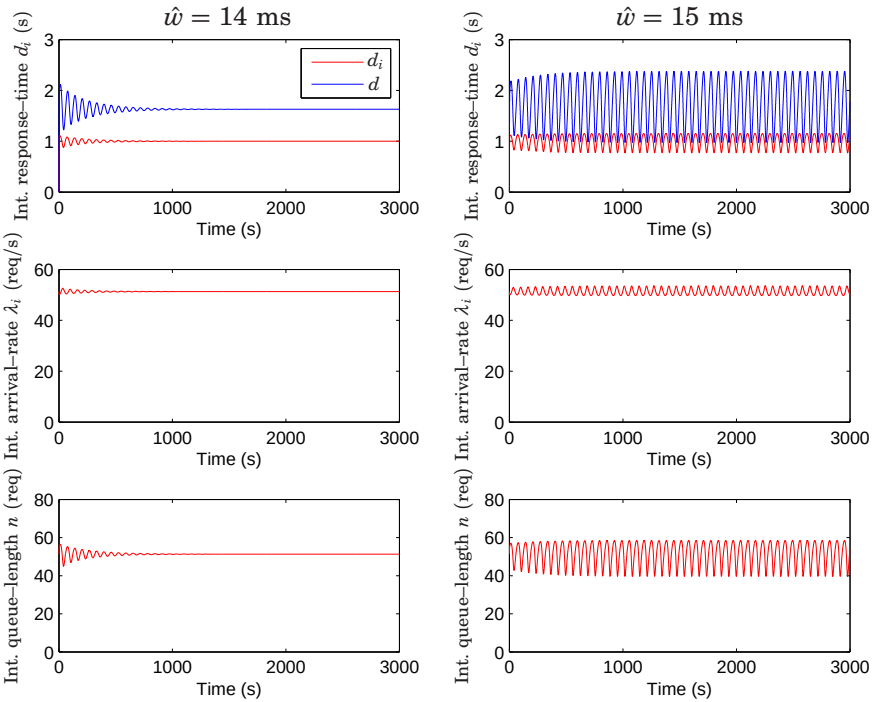

Figure 5.15 Simulation results presenting the internal measured response–time d_i, the internally measured arrival rate, and the internally measured queue length when utilizing both feedback and feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller was 14 ms in the simulation results presented on the left–hand side, and 15 ms in the simulation results presented on the right–hand side.

Summary for the Verification by Simulations

As the analysis suggested the systems becomes unstable if the value of the estimated required work, $\hat{w}$, is chosen too high. In the saturation where only feed–forward is applied, the simulations have shown that the number of requests in the server will go towards the maximum value, when $\hat{w}$ increases above the stability limit. When both feedback and feed–forward control is applied, all variables start to oscillate. The time period and amplitude was approximately as predicted by the analysis.

5.5 Validation by Experiments

The controllers were implemented on the testbed described in Chapter 3. The condition of all the experiments were as described in the beginning of the chapter (listed in Table 5.2). A saturation was imposed to limit the control signal p_r between 0.01 and 0.89. (10% of the CPU capacity was reserved to the operating system and other tasks, and both the secondary task and the web server were guaranteed at least 1% of the CPU capacity). The system was tested with two control–configurations; with feed–forward control only and with the combined feedback and feed–forward controller.

Feed–Forward Control

Figures 5.16, 5.17, and 5.18 illustrate the results of four experiments where only the feed–forward controller was utilized. Note that the feed–forward controller was first included after 300 s in the experiment with $\hat{w}$=8 ms (the lower right part of the figures). The initial conditions for the experiments were not the same, therefore only the steady–state conditions can be compared. For the case of $\hat{w}$=7 ms (in the lower left corner), the system was stable for a while, but after around 2500 s the system suddenly became unstable. This suggests that the stability limit for the actual system is around 7 ms, which corresponds well with the theoretical analysis, which predicted the stability limit to 7.281 ms.

Feedback and Feed–Forward Control

Figures 5.19, 5.20, and 5.21 illustrate the results of four experiments where both the feedback and the feed–forward controllers were active. Note that the initial conditions for the experiments were not the same, therefore only the steady–state conditions can be compared.

The figures indicate that the system remained stable and well behaved for low values of $\hat{w}$, but certain signs of instability were observed already at $\hat{w} = 6$ ms. Stable oscillations were observed at $\hat{w} = 8$ ms. Here, the experimental results deviate from the analysis, which predicted instability for $\hat{w} > 14.06$ ms.

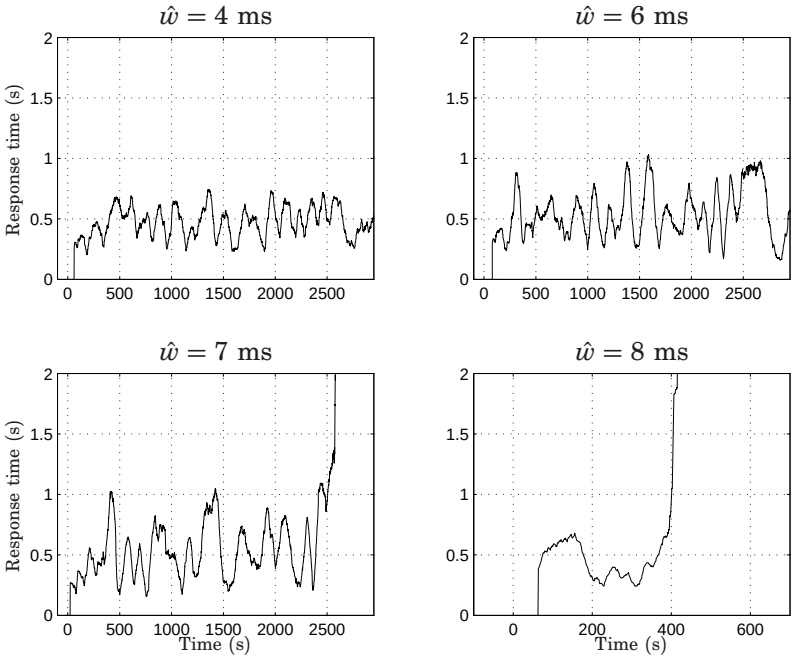

Figure 5.16 Experimental results showing the internal measured response–time d_i when utilizing only feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 4 ms and 8 ms.

Summary for the Validation by Experiments

Both the analysis and the simulations suggested that if the value of the estimated required work, $\hat{w}$, is chosen too high, the systems will become unstable. This was confirmed by the experiments. In the saturation where only feed–forward is applied, the number of requests in the server will increased towards the maximum value. When both feedback and feed–forward control was applied, the system started to oscillate. The limits for stability was somehow lower than predicted by the analysis. This is expected to be related to the model, which does not quite predict the actual levels accurately.

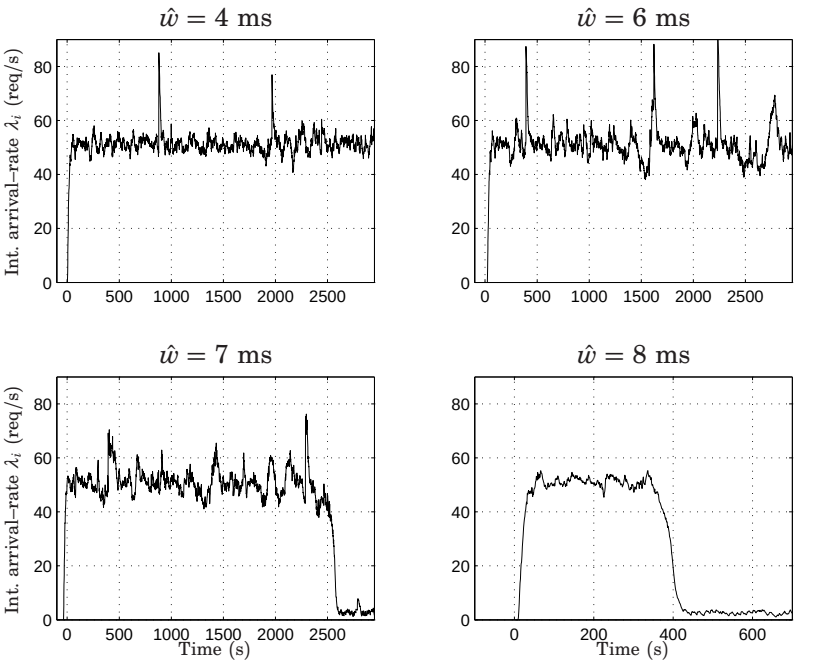

Figure 5.17 Experimental results showing the internal measured arrival–rate λ_i when utilizing only feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 4 ms and 8 ms.

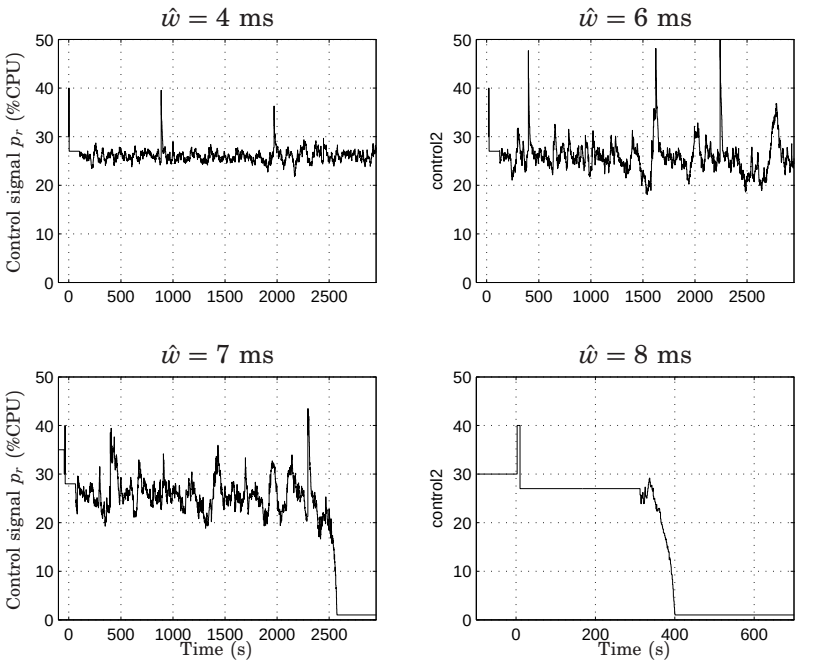

Figure 5.18 Experimental results showing the control signal p_r when utilizing only feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 4 ms and 8 ms.

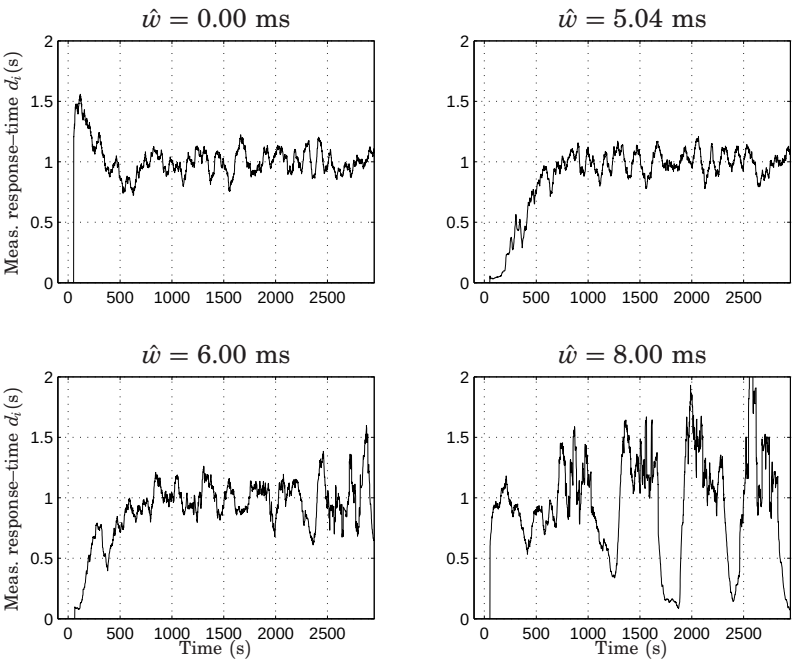

Figure 5.19 Experimental results showing the internal measured response–time d_i when utilizing both feedback and feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 0 ms and 8 ms.

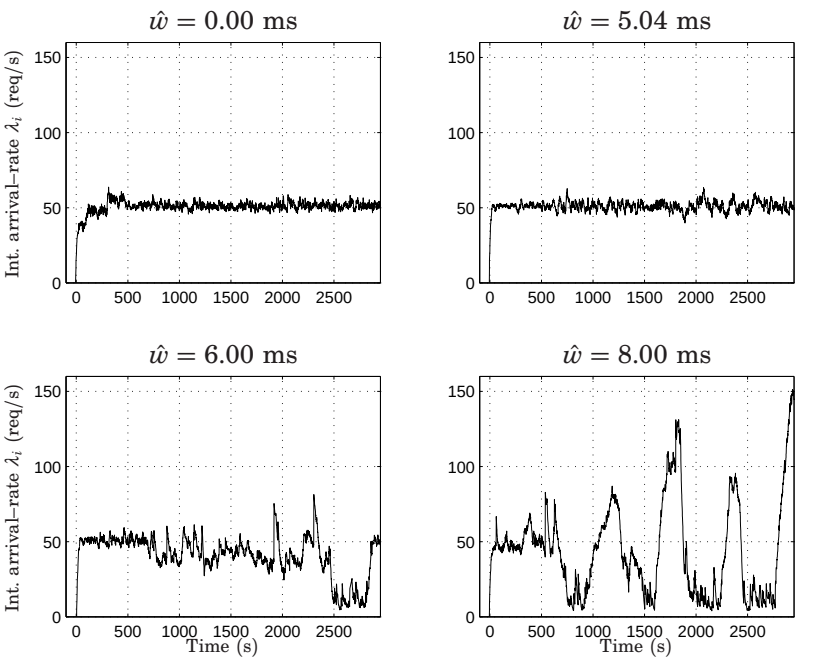

Figure 5.20 Experimental results showing the internal measured arrival–rate λ_i when utilizing both feedback and feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 0 ms and 8 ms.

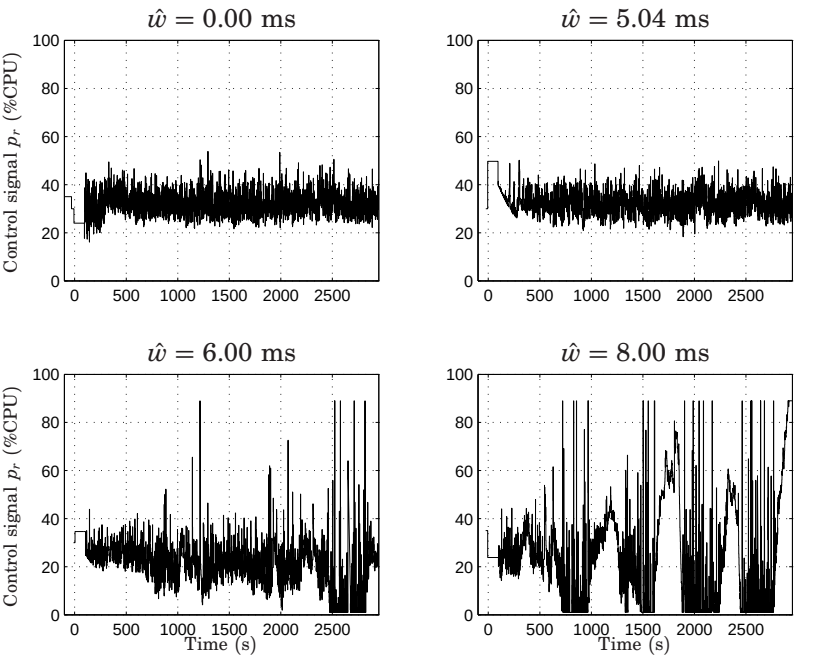

Figure 5.21 Experimental results showing the control signal p_r when utilizing both feedback and feed–forward control. The estimate $\hat{w}$ used in the feed–forward controller is varied between 0 ms and 8 ms.

5.6 Discussions and Conclusions

This chapter demonstrated the danger of neglecting the external buffers during the control design. The measurement of the arrival rate became state dependent when the load was sufficiently high, and the "feed–-forward controller" then introduced a feedback loop. The investigations showed that using a feed–forward strategy alone can lead to instability if the gain of the feed–forward controller is chosen too high. The instability was clearly observed as growing response–times and the control signal going towards zero. If a combination of feed–forward and feedback control was chosen, the system also became unstable if the gain of the feed–forward controller was chosen too high. In this case the instability was manifested as oscillations. The stability conclusions were supported by theoretical analysis, simulations, and experiments, which all showed correspondence in the qualitative conclusions. The stability limit derived from experiments deviated from the stability limit predicted by the analysis, but this does not come as a surprise since the model showed less accurate prediction of the response time (which the feedback was based upon) than for the arrival rate (which the feed–forward controller was based upon). Furthermore, the describing–function analysis is an approximative method, which gives no guarantee of an accurate result.

6

Redesign with Band-Stop Filter

The work presented in this chapter answers an obvious question from Chapter 5: When it is now shown that the feed–forward mechanism introduces risk of instability, what can then be done to avoid this problem? A traditional control–design method is used to expand the robustness towards inaccurate estimates of the parameter $\hat{w}$, and the new controller is verified by both simulations and experiments.

6.1 Redesign of Feed–Forwards

In classical control theory, compensation links are used to shape the open–loop transfer function in order to change the behavior of the closed–loop system, see e.g. [Franklin *et al.*, 1994; Åström and Murray, 2008]. Consider the server system with both feedback and feed–forward control from Chapter 5, and the associated Nyquist plot of Fig. 5.13 on page 95, reprinted in Fig. 6.1 for convenience. It is observed that if the intersection of the Nyquist plot and the real axis could be moved sufficiently towards the origin, the system would no longer be oscillatory. A band–stop filter with a narrow stop–band (notch filter) which stop–frequency ω_s close to the

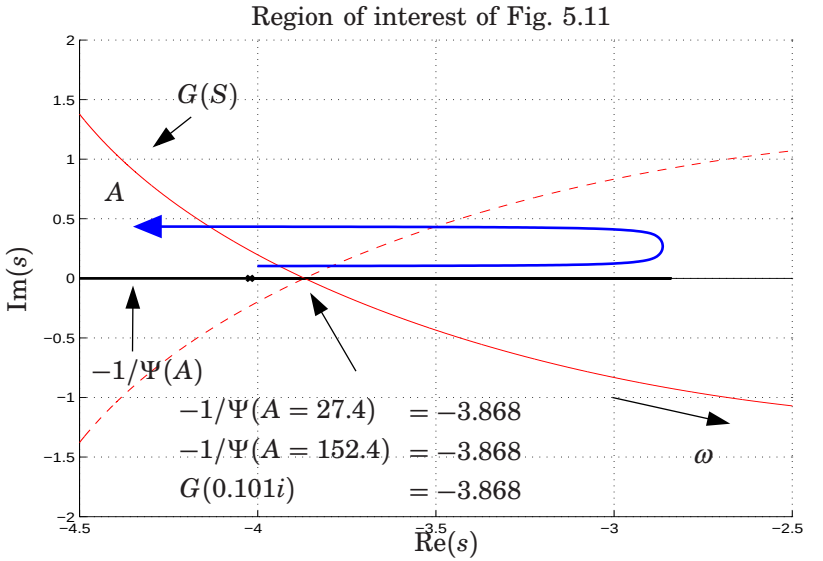

Figure 6.1 Reprint of the close–up of the region of interest of Fig. 5.11 on page 93 for $\hat{w}$=15 ms along with a plot of the negative inverse describing function $(-1/\Psi(A))$.

resonance frequency (ω_s=0.101 rad/s) attains this, without altering the dynamic properties significant for the other frequencies. The filter should be implemented before or after the linear system, as indicated by the *desirable compensation* box in Fig. 6.2. In traditional control systems where the nonlinearity is located in the actuator or the sensor, this causes no problems, since the signals U and Y are accessible directly in the controller. In the case investigated here, the nonlinearity represents a relation between two internal variables, and it is thus not feasible to insert a compensation link at the desired location. Instead, the compensation has to be placed at the same location as the controller, as indicated by the *feasible compensations* boxes.

Because the feasible compensations $G_{fb}(s)$ and $G_{ff}(s)$ both are included in feedback loops, the system cannot be redrawn to a form where the compensations are placed in series with the original system, as with the desired compensation. Therefore, the traditional loop–shaping method cannot be applied here. However, the idea of using a band–stop filter to avoid excitation of the resonance frequency can still be applied. Consider the systems represented in Fig. 6.3, where both are consisting of a transfer function and an ideal band–stop filter. The ideal band stop filter has the characteristic that its amplification is zero at the stop–frequency ω_s, and

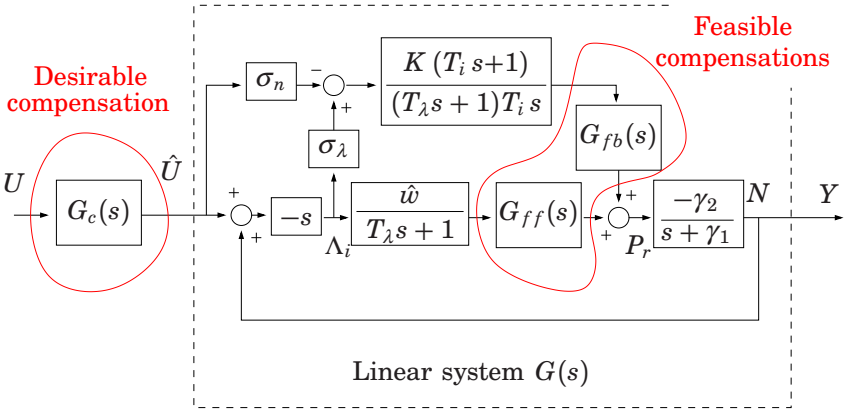

Figure 6.2 Expansion of the block diagram of the linear part of the system, (see also Fig. 5.9), with both desirable compensation and feasible compensations included.

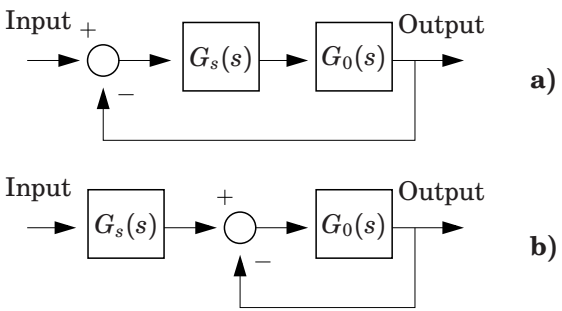

Figure 6.3 Two feedback connections involving a ideal band–stop filter. The two systems holds the same dynamical properties.

one elsewhere. As indicated by Table 6.2, the two feedback systems have the same dynamical properties, so the desired effect of placing a band–stop filter at the input of the linear system of Fig. 6.4 can be obtained by using the same filter inside the controller (marked as *feasible compensations* in the figure). Investigation shows that the best results are obtained using the band–stop filter at the feed–forward controller ($P_{ff}(s)$), and not at the feedback $P_{fb}(s)$. Fig. 6.4 shows the block diagram of the system when the feed–forward loop is broken, and p_{ff} and λ_i are seen as input and output of a linear system, respectively.

Table 6.2 Properties of the systems illustrated in Fig. 6.3.

Frequency	Ideal band-stop filter	Closed loop transfer function of system a) in Fig. 6.3	Closed loop transfer function of system b) in Fig. 6.3
	$G_s(s)$	$\frac{G_0(s)\,G_s(s)}{1+G_0(s)\,G_s(s)}$	$\frac{G_0(s)\,G_s(s)}{1+G_0(s)}$
$s=\omega i, \omega=\omega_s$	0	0	0
$s=\omega i, \omega\neq\omega_s$	1	$\frac{G_0(\omega i)}{1+G_0(\omega i)}$	$\frac{G_0(\omega i)}{1+G_0(\omega i)}$

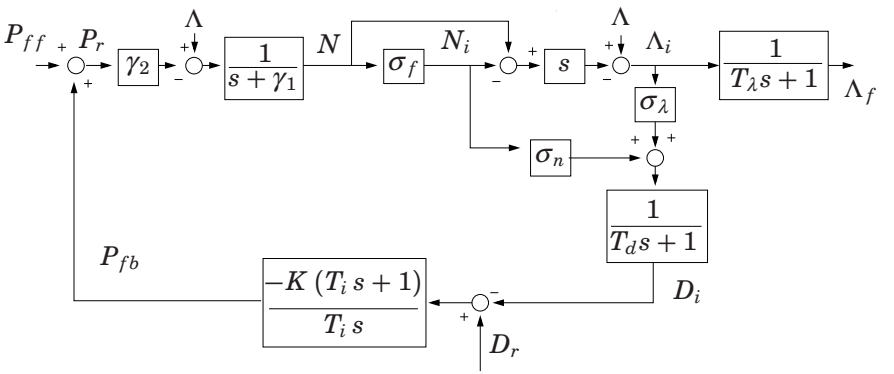

Figure 6.4 Block diagram of the server system, where the feed–forward loop is broken, such that the feed–forward control–signal $p_{ff}$is the input and the measured arrival rate λ_i is the output.

A second–order band–stop filter can be realized by

$$G_{ff}(s) \quad = \quad \frac{s^2+\omega_s^2}{s^2+2\zeta\omega_s s+\omega_s^2} \tag{6.1}$$

where the parameters ω_s and ζ are the stop frequency and the damping factor (zero for non–damped and one for maximal damping), respectively. The choice of the parameter ζ reflects how small the stop band becomes. If the resonance frequency is estimated accurately, a narrow stop–band can be used by choosing a damping factor in the range of 0.6-0.7. If the

estimate is not that accurate, the stop–band can be expanded by choosing a value of ζ closer to one.

The parameter ω_s is chosen as the estimated resonance frequency $\hat{\omega}_c$, which can be estimated by running the system in the unstable operation, and then determine the frequency of the oscillations. A more automated method which does not require unstable operation is relay–feedback which is also used in auto–tuning of PID controllers to identify the same information as here [Åström and Hägglund, 1984]. It should be noted that the parameter $\hat{\omega}_c$ does not only depend on the client properties, but also on the operating point of the system, and thus on numerous factors, such as the control–signal level, the arrival rate, and other disturbances.

6.2 Stability Analysis

Under the assumptions used in the previous chapter, the Nyquist diagrams of both the compensated system and uncompensated systems are illustrated in Figs. 6.5 and 6.6. The first figure represents the case where only feed–forward control is applied ($K = 0$), and where the uncompensated system is on the stability limit (the Nyquist curve intersects with the point -1). It is observed that the compensated system remains stable, and can remain stable for an increase of $\hat{w}$ of approximately 10%.

Figure 6.6 represents the case where feedback is used together with feed–forward control ($K = 0.1$), and where the uncompensated system is on the stability limit (the Nyquist curve intersects with the point -1). It is observed that the compensated system remains stable, and can remain stable for an increase of $\hat{w}$ of 85%.

Evaluation of the closed–loop poles of the entire system (including both feed–forward, feedback, and compensation) reveals the stability results presented in Fig. 6.7. The lines represent the transitions between stability and instability in the parameter plane. Remember that the feed–forward requires the choice of two parameters when the band–stop filter is included. The area above a line represents unstable operation, while the area below a line represents stable operation. It can be seen that the stability limit is pushed upwards when the band–stop filter is included. The effect is most significant when the estimated resonance frequency ($\hat{\omega}_c$) is close to the real resonance frequency (ω_c), and the effect diminishes as the error becomes more pronounced. An important observation is, that the stability properties of the compensated system is always better than for the uncompensated system. That is, from a stability point–of–view, there are no drawbacks from implementing the band–stop filter.

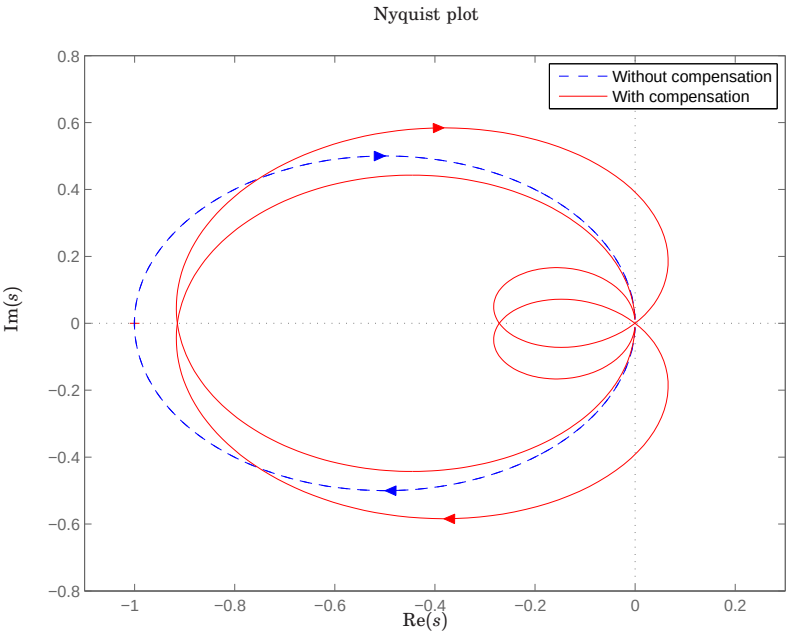

Figure 6.5 Nyquist diagram for the system with feed–forward control. The estimated required work is $\hat{w}$=7.2833 ms, so the uncompensated system is on the stability limit, while the compensated system has an amplification margin of 1.0936.

6.3 Verification by Simulation

The simulation model described in Chapters 4 and 5 was expanded to include the band–stop filter. The operating point was as described in the beginning of Chapter 4 (also listed in Table 5.2). Simulations were performed for three configurations; without the feed–forward (PI control alone), with feedback and uncompensated feed–forward control (similar configuration as in Chapter 5), and with feedback and band–stop compensated feed–forward. The whole point of including the compensation is to expand the robustness towards inaccurate estimates of $\bar{w}$, so the system was simulated with different values of $\hat{w}$. All simulations showed the response to a step in the arrival rate from 36.3 req/s to 51.3 req/s at time t=6000 s. The parameter $\hat{\omega}_c$ was chosen to 0.101 rad/s, and thus, represents a quite accurate estimate of ω_c.

The left of Fig. 6.8 shows how the system should behave if the estimated parameter $\hat{w}$ was estimated accurately ($\hat{w} = \bar{w}$). Here, the effect

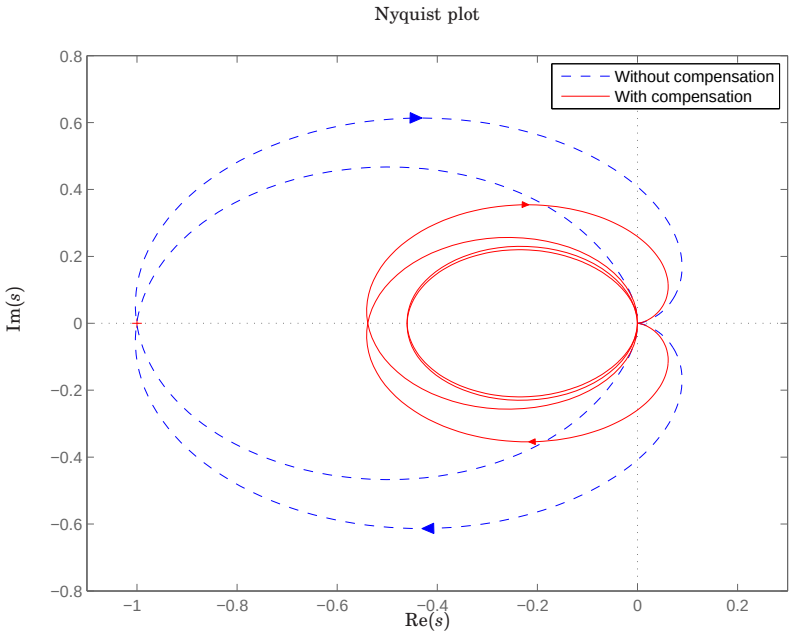

Figure 6.6 Nyquist diagram for the system with feedback control and feed–forward control. The estimated required work is $\hat{w}$=14.61 ms, so the uncompensated system is on the stability limit, while the compensated system has an amplification margin of 1.856.

of the feed–forward clearly shows, as the transient of the system without any feed–forward had a large over–shoot. Since the estimated value of $\bar{w}$ was accurate, the uncompensated system performed very well, while the compensated system showed some degradations in the performance. The degradation occurred because information about the change in the system was removed by the filter to some degree, and thus, the feed–forward could not react as fast.

The left of Fig. 6.8 shows how the system behaved when some inaccuracy in the estimate of $\bar{w}$ was included. Now the systems with feed–forward overcompensated for the disturbance, and therefore, the response time actually decreased for a while when the arrival rate increased. The difference between the compensated system and the uncompensated system was not significant.

The left of Fig. 6.9 shows the situation where $\bar{w}$ is very poorly estimated. When $\hat{w}$=14 ms, $\hat{w}$ was just below the stability limit. The simulation showed no significant deviation between the compensated system

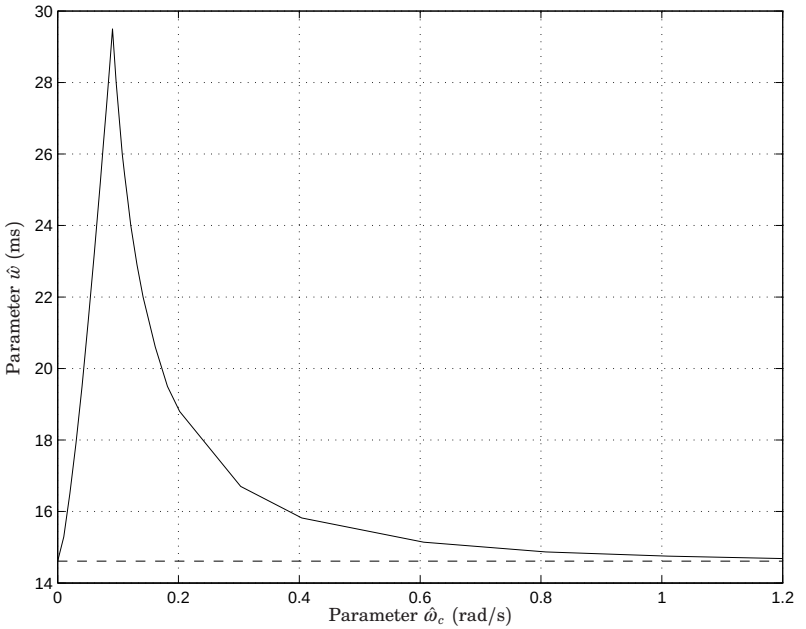

Figure 6.7 Numerical investigation of the stability region for $\zeta = 1$. The full blue line represent the stability limit for the system when the band–stop filter is applied, and the dashed line represent the system without the band–stop filter. Configurations above a stability limit are stable (all poles have negative real parts) while configurations below a stability limit are unstable (one or more poles with positive real parts).

and the uncompensated system, but it can very well be argued that the system performed better without any feed–forward at all.

The right of Fig. 6.9 shows the situation where $\bar{w}$ is so poorly estimated that the uncompensated system enters instability. Here, the compensated system remains stable, which was the main purpose of the compensation. Still, the system without any feed–forward shows a more acceptable performance than the compensated feed–forward controller.

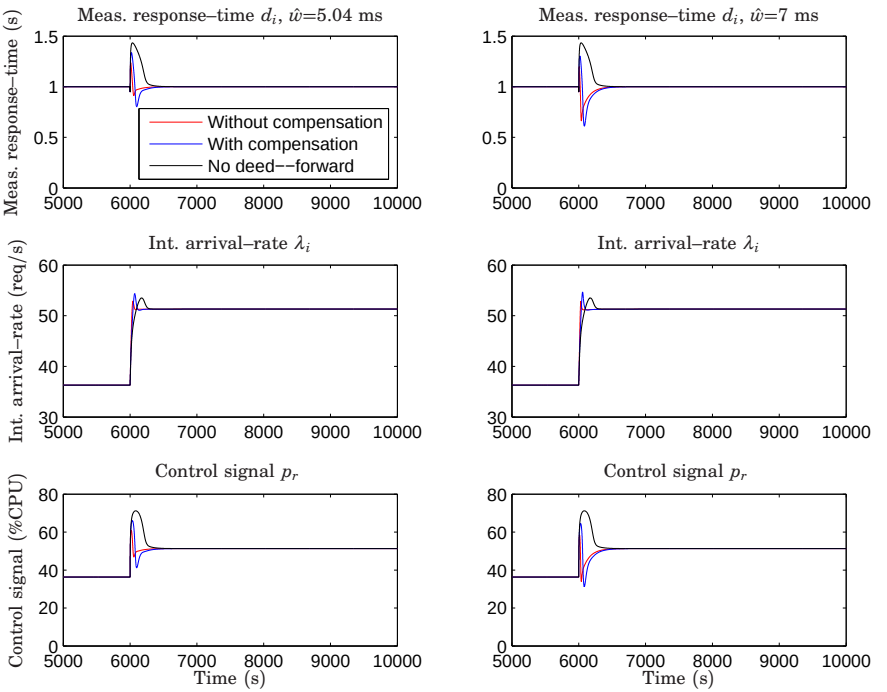

Figure 6.8 Simulation results showing the transient behavior during changing arrival rate for the three cases; when no feed–forward is applied, when the feed–forward is applied without band–stop filter, and when the feed–forward is applied with the band–stop filter.

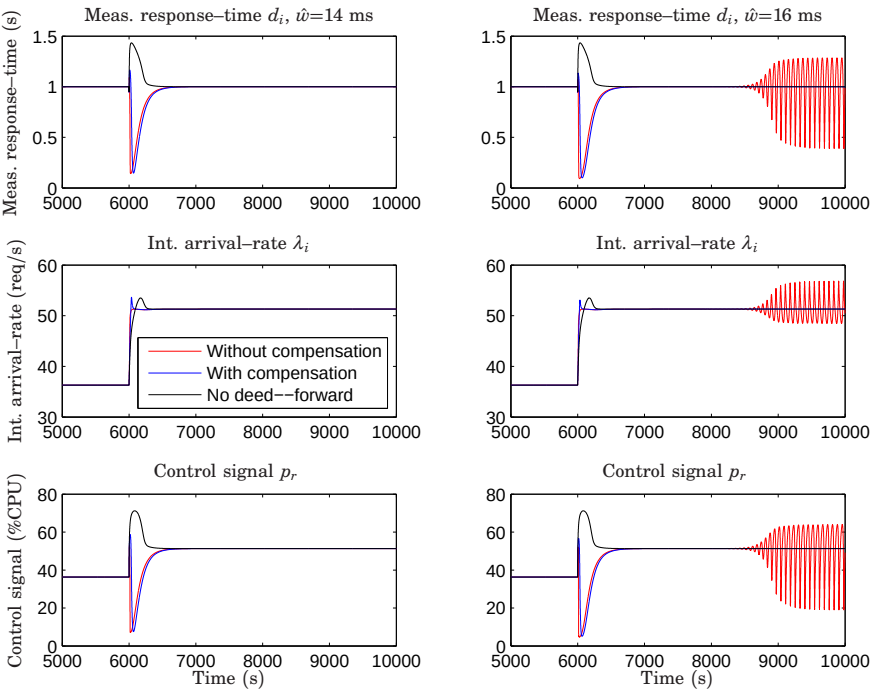

Figure 6.9 Simulation results showing the transient behavior during changing arrival rate for the three cases; when no feed–forward is applied, when the feed–forward is applied without band–stop filter, and when the feed–forward is applied with the band–stop filter.

6.4 Validation by Experiments

The band–stop filter was implemented on the testbed described in Chapter 3 along with the controllers described in Chapter 5. The condition of all the experiments were as described in the beginning of Chapter 5 (also listed in Table 5.2), and only the case of the combined feedback and feed–forward controller was investigated. The system was tested for different values of $\hat{w}$ to investigate if the compensation link expanded the robustness as suggested by the analysis, and also different values of $\hat{\omega}_c$ were tested to investigate the robustness towards inaccurate estimates of ω_c. Figs. 6.10 - 6.12 show the results of six experiments, all in the same order. The upper left of Figs. 6.10 - 6.12 repeat the results of the uncompensated system from Chapter 5. The system was unstable for a highly over–estimated $\bar{w}$. Under the same conditions, the compensated system remained stable (the three experiments shown to the right of the figure), even for different values of $\hat{\omega}_c$. This supports the analysis presented in Fig. 6.7, which predicted that the compensated system would remain stable for values of $\hat{w}$ slightly above the stability limit of the uncompensated system, even for relative large faults in the estimate of ω_c.

The middle left parts of Figs. 6.10 - 6.12 show that the compensated system remained stable when $\hat{w}$ was increased further, but as expected from the analysis, a limit existed, where even the compensated system became unstable. This is seen in the bottom left parts of Figs. 6.10 - 6.12, where $\hat{w}$ was so high that the system entered oscillations.

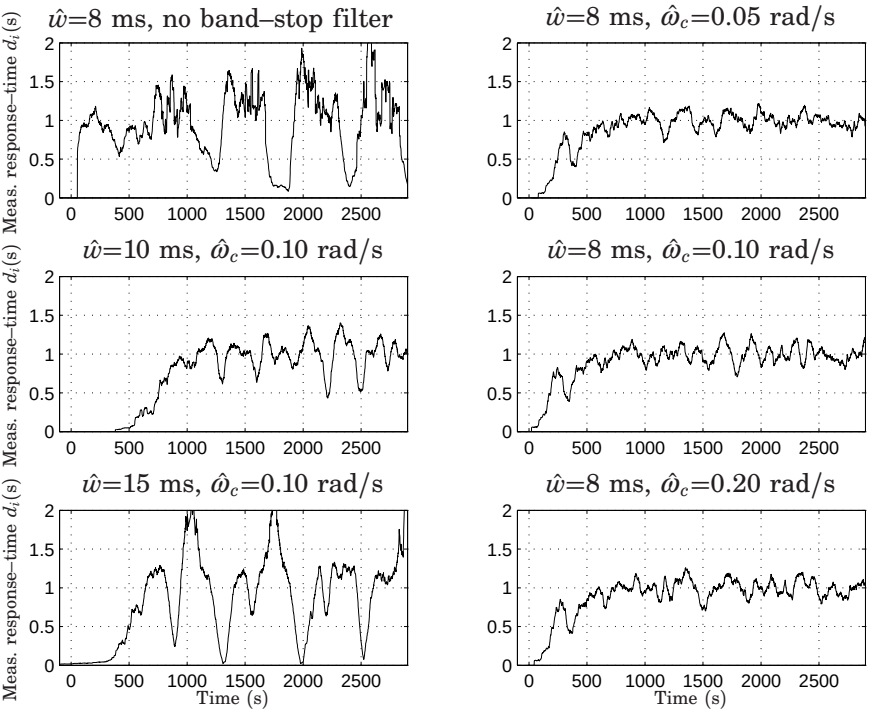

Figure 6.10 Experimental results showing the internal measured response–time d_i when utilizing the band–stop filter. The upper left figure shows the results when the band–stop filter is not applied, and represents the same experiment as in the lower left of Fig. 5.19.

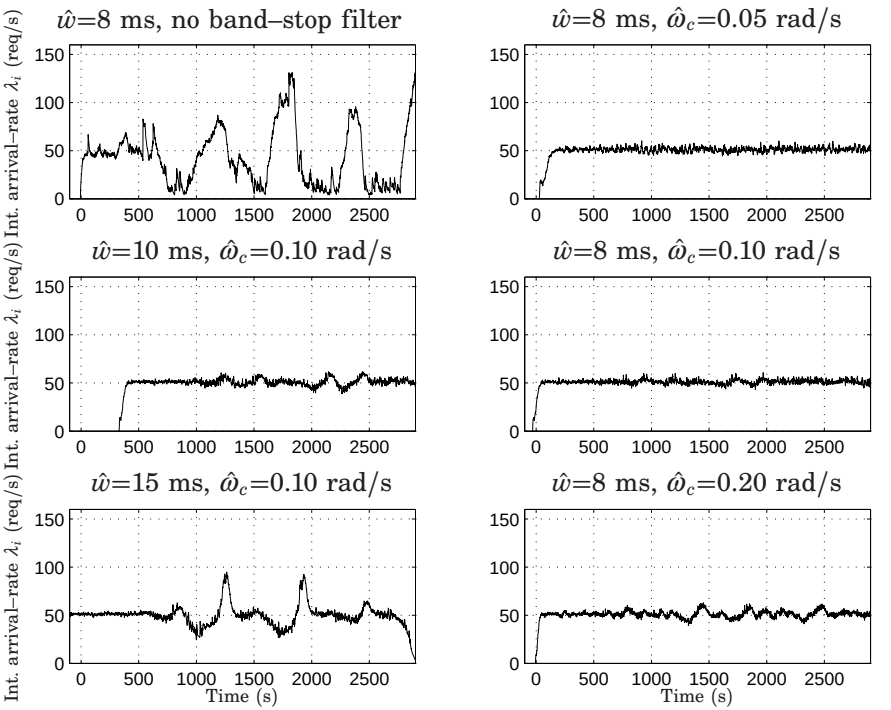

Figure 6.11 Experimental results showing the internal measured arrival–rate λ_i when utilizing the band–stop filter. The upper left figure shows the results when the band–stop filter is not applied, and represents the same experiment as in the lower left of Fig. 5.20.

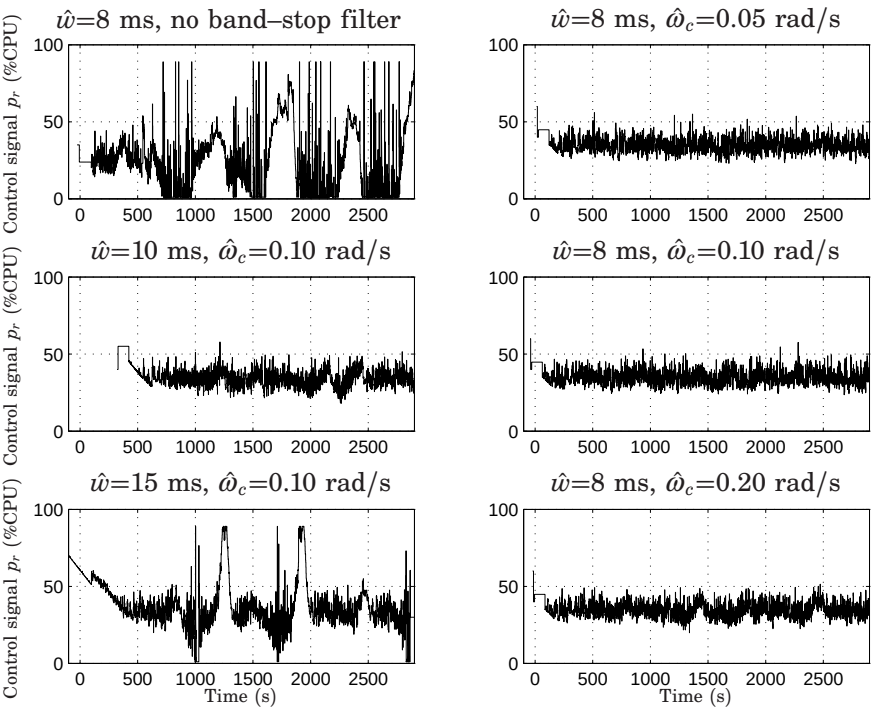

Figure 6.12 Experimental results showing the control signal p_r when utilizing the band–stop filter. The upper left figure shows the results when the band–stop filter is not applied, and represents the same experiment as in the lower left of Fig. 5.21.

6.5 Discussions and Conclusions

This chapter has presented a compensation method to expand the stability range of the feedback/feed–forward configuration presented in Chapter 5, which has showed to become unstable when a certain parameter is chosen too large. The solution presented in this chapter introduces a new parameter ($\hat{\omega}_c$) which represents the estimate of the frequency of the oscillations that the system will enter, if it becomes unstable. So, basically, the method suggests to improve the robustness towards one estimation error by introducing another. The estimation of $\hat{\omega}_c$ is not as critical as the estimation of $\hat{w}$, since a poorly estimated $\hat{\omega}_c$ will only reduce the effect of the compensation, and the worst scenario is that the the compensated system becomes equal to the uncompensated. However, the more accurately $\hat{\omega}_c$ is estimated, the more the robustness is increased.

The method to find $\hat{\omega}_c$ is not investigated in depth here. Systematic methods do exist, but require retuning on a regular basis, since ω_c depends on numerous factors, such as arrival rate, control signal level, and more.

As with most other systems, this system holds a trade–off between robustness and performance, but in this situation, the performance does not seem to suffer that much from including the compensation. As both the compensated and the uncompensated system generally suffers from reduced transient performance for large estimation errors of $\hat{w}$, alternative methods to estimate $\hat{w}$becomes attractive. The next chapter presents a control scheme where $\hat{w}$is estimated online.

7

Improved Feed–Forward Control by Prediction

In this chapter an existing feed–forward strategy is expanded in order to self–adapt to unknown, and possibly slowly varying, parameters. The objective of control is the response time, and the actuation method is the same type of virtualization as assumed in Chapters 4 and 5. The proposed method is compared to the original method and to a queuing–theory based method similar to the one described in Chapter 5. The focus of this chapter is the improvement of the feed–forward mechanism, and the feedback design is thus not treated in greater detail. The material presented in this chapter is based on the work presented in [Kjær et al., 2007; Kjær et al., 2008; Kjær et al., 2009]

As described in Section 3.3, one view is to consider the control of a web server as mainly a disturbance–rejection problem. If the disturbances are not measurable, their effect can only be detected after they have propagated through the system to the outputs. A more desirable behavior can be obtained if the disturbances are detected and compensated for at an early stage. By measuring the disturbances, feed–forward mechanisms are feasible. Here the problem is that a fairly accurate model of the relationship between the disturbance and the output is needed. In this chapter such a model is developed and verified towards other solutions.

7.1 Control Design

The objective of the controller is to fulfill the SLA of the application, that is to keep the average response–time below a reference value, d_r, at the same time as the reserved share of CPU capacity for the application, p_r, is minimized. In steady–state, this can be obtained by feedback mechanisms including integral effects (such as integral controllers and step controllers), which are tuned conservatively to avoid oscillations. However, when changes in the workload occur, this solution is far too slow and a more advanced adjustment of the control signal is necessary. Therefore, it is an objective to remove effects of load–changes as fast as possible.

From a control–theoretic perspective, the system has one control input, the reserved CPU capacity, p_r, and two disturbance inputs, the arrival times of requests, a, and the required work, w. The control objective is to alter p_r in order to maintain the output, *i.e.* the response time, d, close to the reference value d_r despite the behavior of the two disturbances. The interaction between the controller and the server system is illustrated in Fig. 7.1. It is assumed that the arrival times of requests, a, the instantaneous number of requests in the server, $\tilde{n}$, and the response times, d, are available for measurements. Also, it is assumed that the reserved CPU capacity, p_r, can be set online at certain time intervals. The controller will have a larger potential to handle changes in the arrival rate than in the required–work distribution as the controller has direct access to the behavior of the arrivals through measurements. Changes in the required–work distribution are much harder to detect since they are seldom directly measurable, and often the changes will have to propagate to the response times before being recognized.

Figure 7.2 illustrates a control scenario consisting of a feedback controller to assure convergence of the response times and a feed–forward controller to assure fast response to changes in the measured disturbance (the arrival rate). This combination of feedback and feed–forward control has been used extensively within the field of automatic control and also in the more specific case of web–server control [Liu *et al.*, 2006; Lu *et al.*, 2003; Henriksson *et al.*, 2004]. Accurate models of the system can be hard to find, so feedback is an attractive solution to ensure that the CPU resources adapt as deviations between the measured response time and the desired response time are observed. Changes in the arrival rate will require changes in the allocated CPU–capacity if the response times are to remain the same. The feed–forward controller uses information about the arrival rate and provides some suggestion on how to compensate for the change. This suggestion is usually based on a model, and as stated earlier, accurate models are hard to obtain. The resulting compensation may therefore be inaccurate, but here, the feedback controller will com-

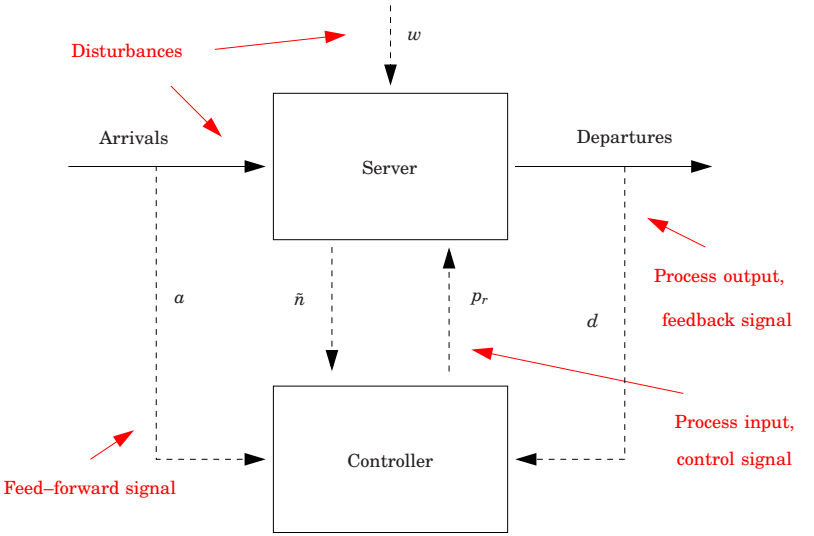

Figure 7.1 An illustration of the control system. Full–line arrows indicate how requests arrive and depart from the server. Dashed arrows indicate signals sent between the server and the controller.

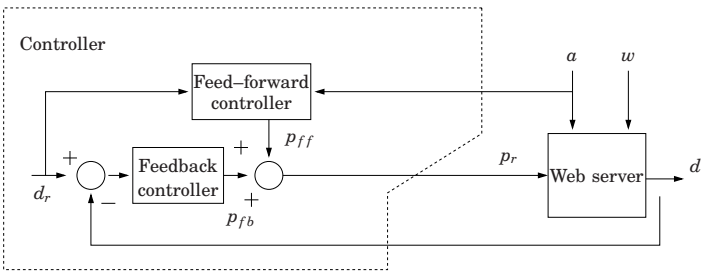

Figure 7.2 Block diagram of a combined feedback, feed–forward setup.

pensate. The feedback mechanism reacts when an error in the response times has been observed, and action is taken rather conservatively to avoid instability. This means that the more inaccurate the feed–forward compensation is, the longer time is required for the controller to compensate for the change in the arrival rate, and the deviation of the response time may also be larger in amplitude. It is therefore of high relevance to find an efficient and accurate method to derive a compensation signal for the feed–forward controller.

Queuing–theoretic expressions have been used for feed–forward signals [Liu *et al.*, 2006; Lu *et al.*, 2003]. These expressions are based on the

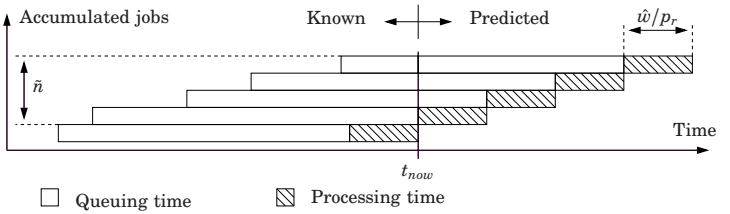

Figure 7.3 Accumulated jobs in a single server queue.

measurement of the arrival times and information on the average required work. Another feed–forward strategy is based on a prediction method, presented in [Henriksson *et al.*, 2004]. This method incorporates information of the instantaneous number of jobs present in the system ($\tilde{n}$) and also an estimate of the average required work. All these feed–forward controllers are based on an estimate of the required work, which is obtained offline. That is, before the system is set into operation, the server is exposed to some traffic (often with low arrival rate), and the average required work is then estimated from measured data. This estimate is then inserted into the controller, and the system is put into real operation. As the required work is not necessarily constant, the estimated required work may easily deviate from the real value, and the feed–forward signal can easily be inaccurate. To avoid this situation, the required work can be estimated online. In the following, a feed–forward model is derived. It is based on work presented in [Henriksson *et al.*, 2004], where the required work is estimated offline.

Prediction Model

The model is based on a prediction approach similar to the one derived by [Henriksson *et al.*, 2004] as illustrated in Fig. 7.3.

The server is modeled as a single–server system where requests are placed in an infinite queue and then processed in a First-In-First-Out fashion. The time to process a job is modeled as being inversely proportional to the reserved share of the CPU, p_r, because the CPU is the most dominant factor limiting the system, which is a main assumption stated in Section 3.1.

Consider the case where a request leaves the server at time t_{now} leaving $\tilde{n}$ remaining jobs as in Fig. 7.3. The area to the left of t_{now} represents the time that the present jobs have spent in the server, which is known from measurements in the server. The area to the right of t_{now} represents the unknown future, which can only be predicted. By assuming that all jobs will have the same required work, $\hat{w}$, and that p_r remains constant, a

prediction of the average response time $\hat{d}$ is given by

$$\hat{d} = \frac{1}{\tilde{n}} \sum_i (t_{now} - a_i) + \hat{w} \frac{(\tilde{n} + 1)}{2p_r} \tag{7.1}$$

where a is the arrival time of request i. The first term on the right hand side represents the known area of the figure, and the second term represents the prediction. The arrival times and the required work are treated as disturbances, which only means that they are quantities not affectable by the operator or by the computer system itself. Of the disturbances, only the arrival times are measurable.

Proposed Prediction Scheme

The prediction model described in Section 7.1 can be useful for online adjustment of the CPU–allocation parameter, p_r, but it relies on several measurements. The instantaneous number of jobs in the server $\tilde{n}$ and their arrival times, a, are quantities often registered by a real server. However, to accurately estimate the average required work, $\hat{w}$, is not always trivial. In a single server with a queue, $\hat{w}$ could be estimated by measuring former service times, corrected with the current value of p_r. However, more complex systems with for example several protocol layers, or for processor sharing systems, this approach is not feasible. The time to process a request depends on other factors, such as the current number of requests. Therefore, to let the control strategy rely on measurements of the required work, will reduce the applicability of the method as the required work is seldom measurable.

In classic linear estimation problems, models are used to estimate non–measured quantities; see for example [Åström and Wittenmark, 1997]. Often, the measurable variables are compared to the estimated values, and a feedback mechanism tries to minimize the estimation error. The response time, d, is considered as a measurable output and the required work, w, as a state to be estimated. In Fig. 7.4 a PI controller is used to correct the estimated required work until the predicted response time matches the measured required work. Equation (7.1) is used as the prediction model. To stress that the estimator does not rely on measurements of w, an artificial variable, z, is imposed to act as the input to the model. The interpretation of this artificial variable is somehow strange. The value of z represents the required work that corresponds to a certain response time, if the required work would be deterministic (as in Fig. 7.3). As the required work is most likely not deterministic, the value of z does not match the average required work $\hat{w}$. However, the two variables are correlated, so a change in $\hat{w}$ will also result in a change in z.

As the real required work cannot be negative, the artificial variable, z, is not allowed to become negative. The prediction model is then given by

$$z_k = \begin{cases} v_k & for \quad v_k > 0 \\ 0 & else \end{cases} \tag{7.2}$$

$$\hat{d}_k = \frac{1}{\tilde{n}_k} \sum_i (t_{now} - a_i) + \frac{(\tilde{n}_k + 1)}{2p_r} z_k \tag{7.3}$$

where v is the control signal from the PI controller. The integral part of the PI controller is implemented with a variable sampling time as the estimation is updated when a request departs from the server which happens sporadically and not periodically. Because of the saturation on the control signal formulated in Eq. (7.2) integrator anti–windup is included as the last term in Eq. (7.4). The PI controller is then given by

$$I_k = I_{k-1} + \frac{h_k K}{T_i}(d_k - \hat{d}_k) + \frac{h_k}{T_a}(v_{k-1} - z_{k-1}) \tag{7.4}$$

$$v_k = K\,(d_k - \hat{d}_k) + I_k \tag{7.5}$$

where T_i and K are controller parameters, and I is the integrated prediction error. The parameters T_i and K are chosen to balance the trade–off of fast convergence and the robustness towards instability. The variable h_k is the time between the previous and the current sampling. Using a varying sampling period in the integrator has earlier been shown to be superior to fixed sample–periods for some event based systems [Årzén, 1999]. The parameter T_a determines the convergence rate of the anti wind–up; see [Åström and Wittenmark, 1997].

The value of z has an interpretation as the required work, which cannot be negative. The anti–windup (the last term of Eq. (7.4)) ensures stability of the integral part in the PI controller when this restriction is imposed on z in Eq. (7.2). The choice of initial value of I is not essential as long as it resembles realistic values of $\hat{w}$. A sound choice is to initiate I to zero, and let the predictor converge before the prediction signal is used (PI control is used alone in the start–up phase).

Equations (7.4) and (7.5) form a general PI controller where the term z_{k-1} is exchanged with the relevant control signal.

Proposed Predictive Feedback Controller (PFB)

The server system suffers from a significant time delay; a change in p_r will only propagate to measurements of the response time $d$after a certain amount of time. In classic control theory, the performance of systems

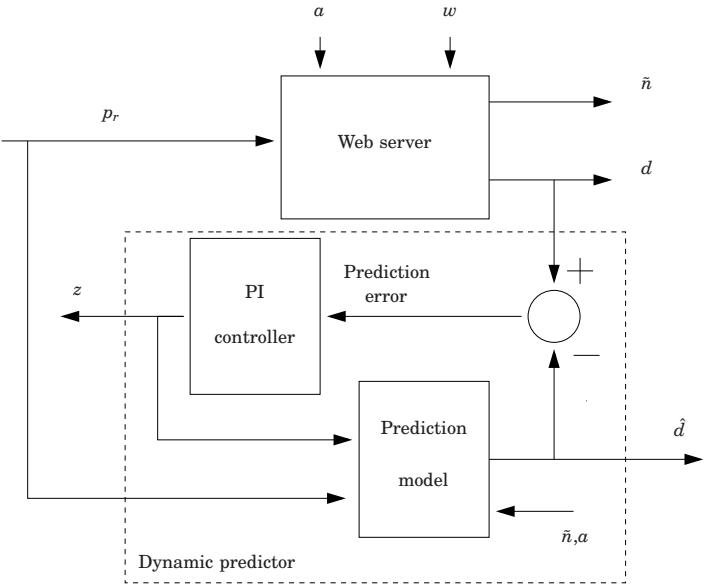

Figure 7.4 Block diagram of predictor. The predictor can be interpreted as an observer with state z.

with delays can be improved by prediction techniques such as the Smith predictor; see [Åström and Hägglund, 2005]. Inspired by this idea, the predicted response time is used as a proportional feedback signal

$$p_p = -K_{pfb}(d_r - \hat{d}) \tag{7.6}$$

to respond to errors that are not yet seen in the response time signal. The prediction signal p_p enters directly on the control input p_r as illustrated in Fig. 7.5, and the parameter K_{pfb} is used to scale the influence of the prediction (the P controller in the figure). In order to handle model errors, a periodic PI–controller from the actual response time is included. The combination of the PI controller, the predictor and the P controller is in the following sections called the *Predictive Feedback Controller* (PFB).

Figure 7.6 illustrates the control structure as a combination of feedback and feed–forward control. Compared to the control structure in Fig. 7.2 the predicted response time is now used as input to the feed–forward.

It is an assumption that the reserved share of CPU capacity, p_r, only can be changed at some fixed time period, T_s. However, the estimation is not restricted by the sampling period. The estimation is updated for each departure ensuring that the response time prediction, $\hat{d}$, and the state, z, always incorporate the newest measurements. The involved signals can be

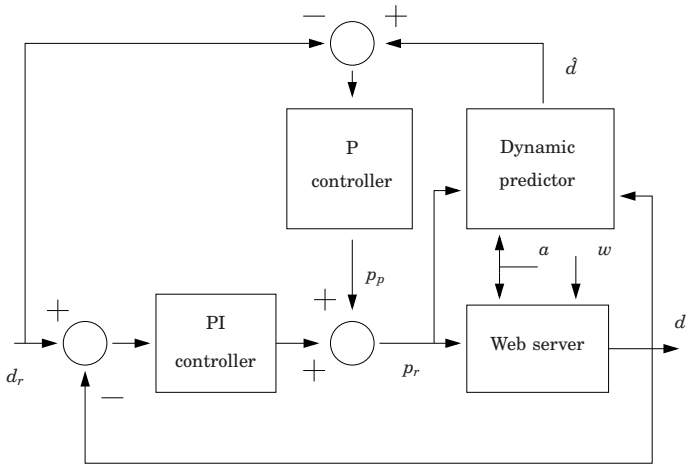

Figure 7.5 Block diagram of the the proposed PFB controller.

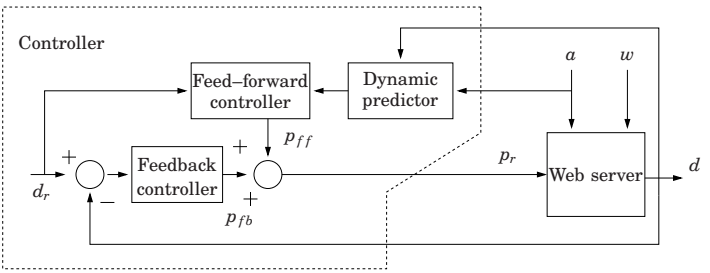

Figure 7.6 Block diagram of the proposed control method, expressed as a combination of feedback, feed–forward setup. Note that the variable $\tilde{n}$ is not included in the diagram.

quite irregular, which can lead to irregular estimation and poor control performance. Therefore, filtering may be required, but this will be discussed in relation to the individual implementations as filtering depends highly on the specific measurements and memory considerations.

Controllers for Comparison

Earlier work presents solutions where feed–forward and feedback control are combined as illustrated in Fig. 7.2 [Liu *et al.*, 2006; Lu *et al.*, 2003; Henriksson *et al.*, 2004]. Similar to the proposed prediction–based controller, the feed-forward uses measurements of the disturbances to change the control signal p_r before a change in the disturbance is seen in the response time, and a PI controller to remove the remaining station-

ary errors. In order to compare the performance of the proposed controller to other feed–forward strategies, the controllers for comparison are used together with the same periodic PI–controller as used for the proposed controller.

As comparison, two feed–forward solutions are presented, which both assume that an estimate of the average required work is available. An often used procedure to obtain the estimate is to measure the response times at a very low arrival rate (offline), as described earlier. Assuming that only one job is present, the response times can be used to estimate the required work. The estimate is then used online at higher arrival rates assuming that the required work will remain unchanged. Obviously, this method is not robust towards changes in the required work as, for example, when the document popularity within a certain web site suddenly changes. Alternatively, response times can be observed at dedicated high–load experiments and an estimate of the average required work can be found by assuming some particular queuing–model, as *e.g.* an $M/M/1$ model. This kind of approach can also lead to inaccurate estimates as the assumed model seldom reflects the reality and it does not handle changes during operation.

Inverse Prediction Feed–Forward (IPFF) This is a slightly modified version of the feed–forward controller presented in [Henriksson *et al.*, 2004]. The feed–forward signal p_{ff}, which enters the control signal according to Fig. 7.2, is found by rearranging Eq. (7.1) so that p_{ff} is the control signal required in order to obtain the desired response time d_r ($\hat{d}$ is exchanged with the desired value d_r). Thereby, the feed-forward is given by

$$p_{ff} = \frac{\tilde{n}+1}{2(d_r - \frac{1}{\tilde{n}}\sum_i (t_{now} - a_i))}\hat{w} \; . \tag{7.7}$$

In [Henriksson *et al.*, 2004], the numerator yields $\tilde{n}$ and not $\tilde{n}+1$.

Queuing–Theoretic Feed–Forward (QFF) The feed–forward part in this controller is based on a queuing–theoretic model. The average response time of a steady $M/G/1\text{-}PS$ system is given by

$$\bar{d} = \bar{x}/(1-\lambda\,\bar{x}) \tag{7.8}$$

where λ is the average arrival rate and $\bar{x}$ is the average service time [Kleinrock, 1967; Noguahi and Oizurnih, 1971]. Equation (7.8), or varieties hereof, have formed the base for other feed–forward designs in the literature, *e.g.* [Liu *et al.*, 2006; Lu *et al.*, 2003]. In the specific case of p_r

being constant we have $\bar{x} = \bar{w}/p_r$. Assuming that $\bar{w}$ is known (and exact) and λ is estimated by some windowing mechanism ($\hat{\lambda}$), a feed–forward signal can be formed as

$$p_{ff} = \bar{w}\,(1 + \hat{\lambda}\,d_r)/d_r \tag{7.9}$$

which enters the control signal as illustrated in Fig. 7.2.

7.2 Verification by Simulations

The proposed PFB–controller is mainly designed to improve the transient performance at workload changes. However, it is also expected to handle the short–term stochastic variations observed in the stead–state situations similar to the controllers for comparison. Simulations of a generalized server system with CPU resource allocation were performed. The simulation program was written in Java and used an event-based simulation kernel.

Steady–state and transient simulations were performed. All steady–state results were evaluated after all transients had been removed. Transient behavior was investigated after convergence to steady–state, and the simulations were allowed to run for a sufficiently long time for the transient to settle.

Simulation model

The server system was modeled as a single server queue with processor–sharing. New requests arrived with an average arrival rate of λ requests per second. The requests had a reserved share of p_r of the CPU capacity. The average required–work was $\bar{w}$. Since the server used processor–sharing, w represents the service time a request would get if it were the only request processed in the system. When several requests are processed at the same time, the CPU capacity is divided equally among the requests.

Both the inter-arrival times and the required work were modeled as second order hyper–exponential distributions (H_2 distribution) in order to model a bursty system. An H_2 distributed variable ψ is with probability β a realization of an exponentially distributed variable with expected value ξ_1, and with probability $(1-\beta)$ a realization of exponential distributed variable with expected value ξ_2. The following parameters were used

$$\beta = (C^2 - 1)/(C^2 + 161) \tag{7.10}$$

$$\xi_1 = 0.1\,\bar{x}\ ,\ \xi_2 = \bar{\psi}(1-\beta)/(1-10\beta) \tag{7.11}$$

where C^2 and $\bar{\psi}$ were the squared variance coefficient and average value of the H_2–distributed sequence, respectively. The value of C^2 was chosen

to be equal for the inter–arrival times and the required work distributions and C^2=5 unless stated differently.

The control parameters for the predictor were chosen as $T_i = 0.0005$, $K = 0.000001$, $T_a = 0.5$. The parameters for the periodic controller were chosen as $K = 0.000014$, $T_i = 0.010$, $T_a = 1010$. The proportional gain of the PFB controller was chosen as $K_{pfb} = 0.2$. The parameters have been found by running simulation tests and adjusting the parameters by hand.

The involved signals can be quite irregular, which can lead to irregular estimation and poor control performance. All the tested periodic PI–controllers use the comparison of the reference and a filtered response time d^p; $d^p_k = (d^p_{k-1} + d^w_k)/2$ where d^w_k is the average response time of the jobs that departed under the interval between sampling $k-1$ and k.

To update the predictor, the predicted response–time is compared to a first order auto–regressive filtered measured response–time with filter constant $\alpha = 0.001$. The auto–regressive filter is implemented as

$$d^f_i = (1-\alpha)\, d^f_{i-1} + \alpha\, d_i \quad , \tag{7.12}$$

where i indicates the departing job number and d_i is the response time of job i.

The response time prediction can also be quite irregular. An obvious idea is to apply a filter directly to the predicted response time $\hat{d}$. This has an undesirable effect due to the nonlinear structure. Linear filtering of the term $1/p_r$ would favor small values of p_r and could lead to wrong predictions. Also, a linear filtering of the term $\sum_i (t_{now} - \lambda_i)$ would weight the jobs that have a long service time over those having a short response time, thus increasing the average prediction. The filtering must therefore be placed with care. The best results have been obtained by simply filtering $\tilde{n}$; $\bar{n}^f_i = 0.999\, \bar{n}^f_{i-1} + 0.001\, \tilde{n}_i$, which is an event–based filter.

The IPFF controller is based on inverse prediction. That is, any response time error is compensated in one update. A similar idea is used in classical minimum–variance control, which is known to have poor robustness properties; see [Åström, 2006]. The result is an undesirable irregular control signal and some filtering is imposed. Applying a filter to the control signal would drive the average control signal off due to the nonlinearity of the fraction in Eq. (7.7). Therefore, the numerator and denominator are filtered separately;

$$P_i = 0.999\, P_{i-1} + 0.001\, (\tilde{n}+1)\, \bar{w} \tag{7.13}$$

$$Q_i = 0.99\, Q_{i-1} + 0.01\, \frac{1}{\tilde{n}} \sum_i (t_{now} - a_i) \tag{7.14}$$

$$p_{ipff} = \frac{P_i}{2(d_r - Q_i)} \tag{7.15}$$

Steady–State Simulations

The traffic load is often described by two quantities; the average arrival rate (λ) and the nominal service rate ($1/\bar{w}$). Often, the traffic is quantified by the offered load, $\rho = \lambda\bar{w}$. Assuming a single server system, a low value of ρ means a lightly loaded system, whereas values close to one mean a heavily loaded system. If ρ exceeds one, the system lacks CPU capacity to serve incoming requests, which means that the system is overloaded.

Performance when varying offered load Fig. 7.7 illustrates the performance metrics when the arrival rate was varied in a range corresponding to $\rho = 0.05 - 0.90$. It indicates that all the controllers managed to keep the average response time near the reference. Remember that the 95% confidence interval was ± 0.1 s, so the response time results show no significant deviation between the controllers.

All controllers performed best when the offered load was relatively high as the loss of CPU capacity, q, then was small. In the simulations, the offline estimate of the required work, $\hat{w}$, used in the IPFF and QFF controllers, corresponded exactly to the average required work $\bar{w}$. The proposed PFB controller estimated this parameter online, but showed no significant degradation in performance because of this. The QFF controller showed a higher variation–cost V_d, which indicates a less smooth response time than the other controllers.

Robustness to changes in the required work On a real web site, it is unrealistic that the average required work, w, will be constant during longer periods since the document popularity is likely to change. Therefore, a control system must be robust to changes in the average required work. Fig. 7.8 presents the performance metrics when the required work was varied in a range corresponding to $\rho = 0.14 - 0.875$. In the simulations, the offline estimate of the required work, $\hat{w}$ was therefore inaccurate.

The results reveal that all the controllers managed to keep the average response time near the reference (the reference was within the confidence interval). Also here, the QFF controller performed rather poorly over the full range since it yields both a higher loss of CPU capacity and also a large variation–cost V_d. Despite the inaccuracy of the offline estimate of the required work, the IPFF controller performed well in steady–state because of the robustness of the PI controller. In general, a PI controller can compensate well in steady–state for well–behaved feed–forward errors. However, poorly designed feed–forward controllers will not always be handled by the PI controller.

As the arrival rate becomes small, the number of measurements available to perform a prediction decreases (the predictions are performed

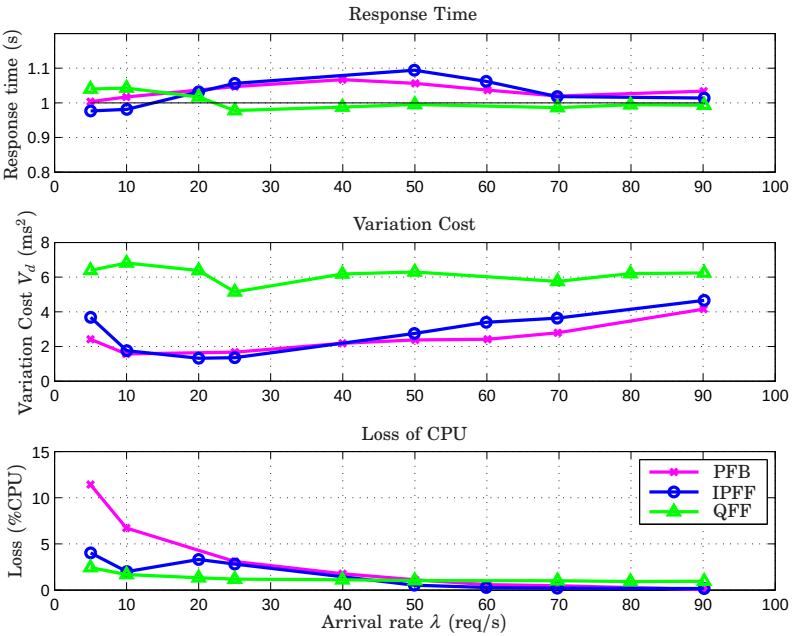

Figure 7.7 Averaged steady–state simulations for different arrival rates λ. C^2=5, $\bar{w}$=0.01 s, $\hat{w}$=0.01 s, T_s=1 s, d_r=1.0 s.

with fixed time–periods). The variance of the prediction increases and thereby generating a more noisy control signal, leading to higher loss of CPUcapacity. This is observed in both Fig. 7.7 and 7.8.

Response Time Reference By ensuring that there always are jobs to process, the loss of CPU capacity will be small. This is obtained by allowing a higher average response time than what may be possible with more allocated resources. Fig. 7.9 shows that there was a limit where it was not worth to increase the response time reference any further. In the given case, response time references above approximately 1 s did not result in any significant reduction in the loss of CPU capacity. It did not matter for the loss of CPU capacity whether there were few (but always some) or many jobs in the system. Thus, a higher response time reference would only generate higher average response time but no improvements. When the response times became small, the risk of an empty system became significant, which introduced loss of CPU capacity. This is clearly indicated for the low response time reference of Fig. 7.9.

Robustness to traffic variance In order to investigate the robustness for more bursty arrival traffic, simulations were performed where the

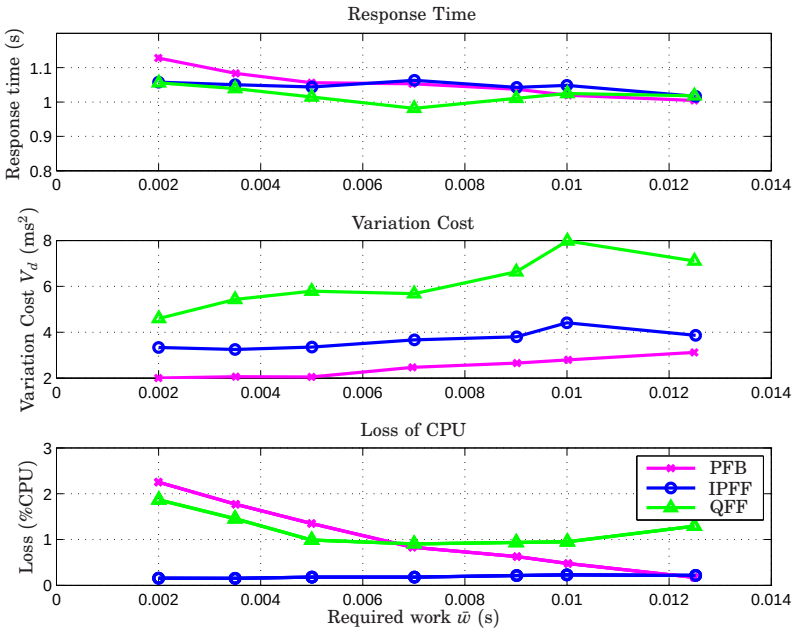

Figure 7.8 Averaged steady–state simulations for different average required work $\bar{w}$. C^2=5, λ=70 req/s, $\hat{w}$=0.01 s, T_s=1 s, d_r=1.0 s.

squared coefficient of variation for the arrival process was varied. Fig. 7.10 shows how the proposed controller, PFB, performed under other types of traffic at different arrival rates. In general, Fig. 7.10 indicates that the controller handles different levels of burstiness of the traffic quite well. The variation–cost, V_d, increased with the burstiness of the arrival process. However, this is an expected result.

Sampling interval Fig. 7.11 shows results that are interesting from an implementation point of view. In some cases, the sampling period for the update of p_r may not be a free choice, but can be restricted by software and hardware limitations. Generally, the results presented in Fig. 7.11 show that the loss of CPU capacity increases with the sampling period. Especially for low traffic, the proposed controller showed degraded performance as the sampling period was increased. The variation–cost, V_d, increased with the sampling period and as a result, higher loss of CPU capacity was observed. For higher arrival rates, the controller showed acceptable performance with sampling periods several times higher than the response time reference. However, when the sampling period was longer than about seven times the response time reference, the performance started to de-

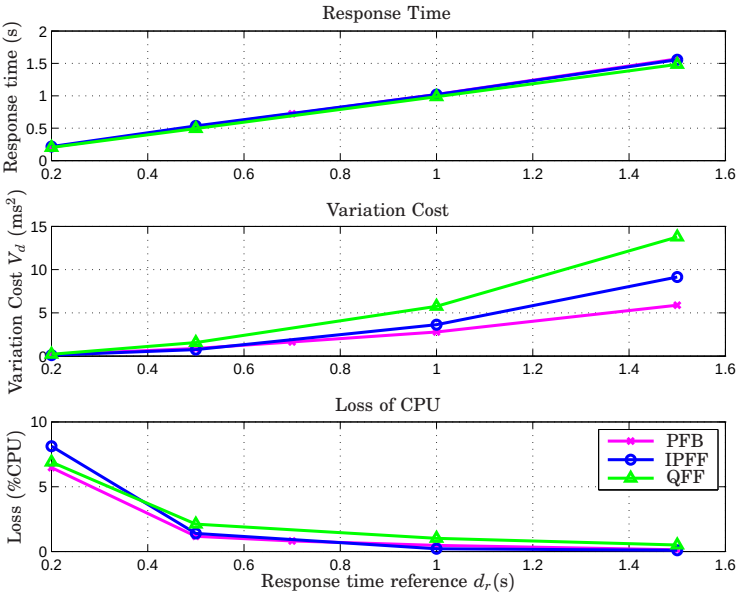

Figure 7.9 Averaged steady–state simulations for different response time references d_r. C^2=5, λ=70 req/s, $\bar{w}$=0.01 s, $\hat{w}$=0.01 s, T_s=1 s.

grade. This can cause a serious constraint on the response time references.

Transient Simulations

One strong argument to use feedback in the control is the robustness towards sudden changes in the environment. Therefore, it is of high importance to also investigate the transient behavior of the controlled system.

Figure 7.14 illustrates how the two main metrics, the response time and the loss of CPU capacity, behave under transients. Using cost–functions averaged over several simulations, improves the accuracy of the results (smaller confidence intervals). Preferably, both cost–functions should be close to zero to yield good performance. The two cost–functions are not necessarily contradictory as the average response time can be held constant, if the CPU capacity is allocated just sufficiently and thus minimizing the loss of CPU capacity. However, the controllers may solve this problem differently, which can be observed in the figure.

The top of Fig. 7.14 illustrates a situation where the average required–work, $\bar{w}$, was suddenly doubled. In the beginning of the simulation, the offered load was relatively low with ρ=0.4 (λ=50 req/s, $\bar{w}$=0.008 s). At

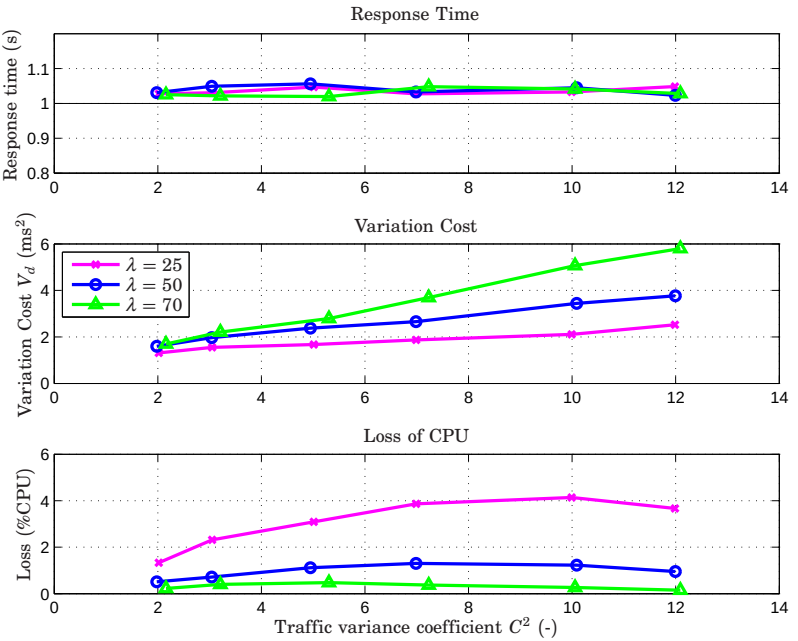

Figure 7.10 Averaged steady–state simulations of the proposed PFB controller for different arrival rates and different variance coefficient C^2. $\bar{w}$=0.01 s, $\hat{w}$=0.01 s, T_s=1 s, d_r=1.0 s.

time t=1000 s the average required–work was doubled ($\bar{w}$=0.016 s), so that the system was exposed to high–load traffic with ρ=0.8. The offline estimated required–work was chosen to be $\hat{w}$=0.01 s to illustrate a slightly inaccurate estimate within the tested range. It can be observed that the proposed PFB controller is superior to the other controllers in the case of changes in the required work $\bar{w}$, as it yields a smaller cost in the response time error and a smaller cost in the loss of CPU capacity. This behavior is expected since the PFB controller estimates the value of $\bar{w}$ online while the two other controllers use offline estimates.

The middle of Fig. 7.14 illustrates that the proposed PFB controller handles a change in the arrival rate better than the IPFF controller. The QFF controller handles the transient with slightly smaller response time errors, but with substantially larger loss of CPU capacity. This behavior can be explained by the behavior also seen in Figs. 7.12 and 7.13, which show a clear over–allocation of CPU resources for the QFF controller. Figure 7.12 illustrates simulation results of a single simulation run, which is quite noisy. To remove noise, 250 similar simulation runs were con-

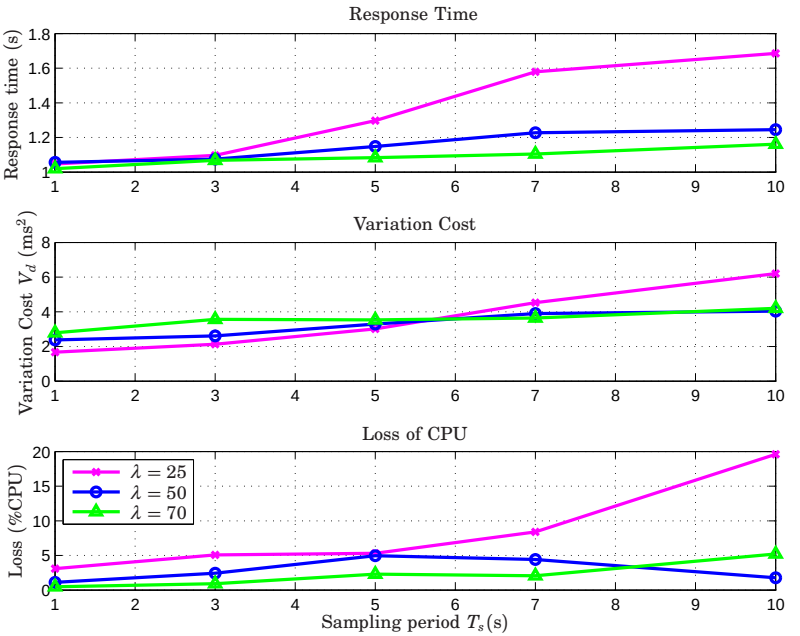

Figure 7.11 Averaged steady–state simulations of the proposed PFB controller for different sampling period T_s; Arrival rate λ=50 req/s; Variance coefficient C^2=5; Average required work $\bar{w}$=0.01 s; Response time reference d_r=1.0 s.

ducted with different seeds in the random–number generator, and the results were averaged. This type of filtering is computationally expensive, but does not influence the dynamics as time–domain filters does. For all simulation runs the system was initially exposed to a low–load traffic with ρ=0.35 (λ=50 req/s, $\bar{w}$=0.007 s). Again, the offline estimated required–work was chosen to be $\hat{w}$ = 0.01 s to illustrate a slightly inaccurate estimate. At time t=1000 s the arrival rate was doubled, so that the system was exposed to high–load traffic, ρ=0.7. Fig. 7.13 also shows that the proposed PFB controller handles the change in the arrival rate with a smaller deviation in the response time but with a slower convergence.

The results presented in the bottom graph of Fig. 7.14 had traffic variance coefficients C^2=1.1 and shows that in the case of lightly bursty traffic, the IPFF controller handles the transient better than the proposed PFB controller. In this situation, the inverse nature of the IPFF controller becomes very beneficial because the model resembles the reality fairly well. The PFB controller does not rely on an inverted model but rather on a feedback mechanism, and does therefore not improve as much from the less bursty traffic.

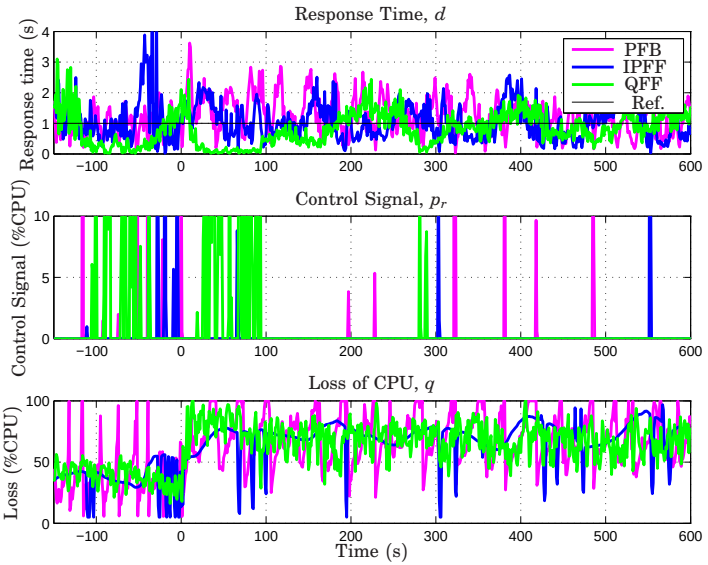

Figure 7.12 Time–domain transient simulation results with changing arrival rate and high traffic burstiness (C^2=5). Each plot represents a single simulation run.

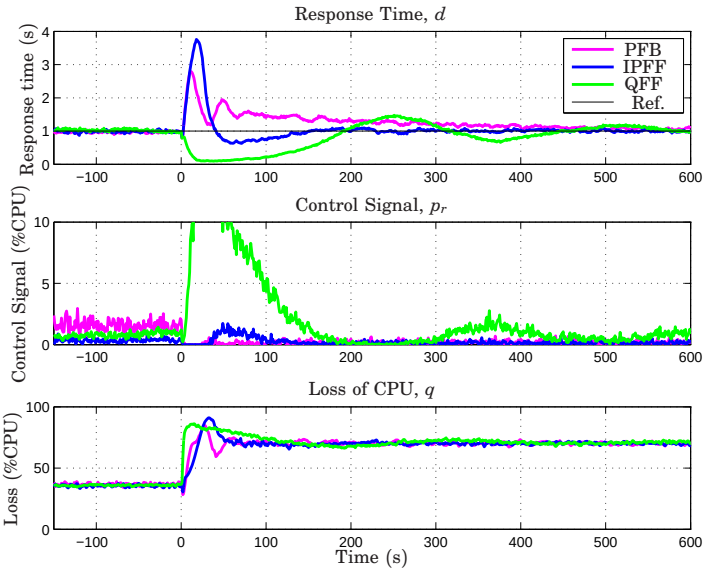

Figure 7.13 Time–domain transient simulation results with changing arrival rate and high traffic burstiness (C^2=5). Each plot represents an average of m=250 simulation runs.

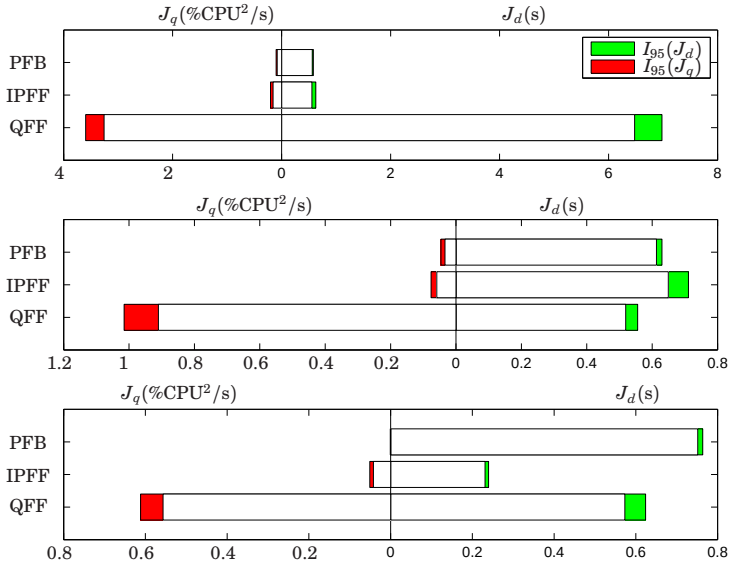

Figure 7.14 Cost–function results over transient period of length t_t (defined in Eqs. (3.1) and (3.2)). Each cost function is an average over m simulation runs. Top: Transient simulations with changing average required–work (m=150, t_t=1700 s). Middle: Transient simulations with changing arrival rate and high traffic burstiness. (C^2=5, m=250, t_t=300 s). Bottom: Transient simulations with changing arrival rate and low traffic burstiness. (C^2=1.1, m=150, t_t=300 s). Generally, the closer a metric is to zero, the better. A controller can show a better performance in one metric but not in the other (*e.g.* the QFF vs. the other controllers in the middle of the figure), but clear improvements in both metrics can also be seen as in the top of the figure where the PFB controller performs better than the others in both metrics.

A general observation from Fig. 7.13 and Fig. 7.14 is that the QFF controller responds poorly to changes. In the case of increasing average required–work the feed–forward controller did not make any difference because it only considered the arrival rate. Therefore, the periodic PI–controller had to handle the change resulting in a large deviation of both the response time and a large loss of CPU capacity. In the cases where the arrival rate changed, the QFF feed–forward over–compensated resulting in a large loss of CPU capacity.

However, for all high–burstiness simulations, the proposed PFB controller showed superior transient response.

7.3 Verification by Experiments

The experiential verification was conducted on the testbed described in Chapter 3. The prediction algorithm was implemented immediately after the measurement update in the `log_transaction`–hook; see page 54. The location of the prediction algorithm was chosen for two reasons. First, it is at this stage the request has been fully processed and the response time can be calculated. Second, if the prediction should impose any overhead, it will not add to the response time of the associated request since an answer has been returned to the client

To avoid unnecessary delays caused by file accessing, data for individual requests were not saved. Instead, the relevant metrics were averaged over a sample interval and saved after the control signal had been set. All results presented here are based on such measurements. The sample interval was 1 s for all experiments. Also, 10% of the CPU capacity was reserved for the basic group (operating system and controller), and the control signal was restricted to be in the interval $1\% - 89\%$. The average required work estimate was set according to the offline experiments to $\hat{w}$=11 ms which approximately corresponded to the estimated required work of the initial traffic; see Table 3.2. The control parameters for the predictor were tuned by hand. All parameters were as for the simulations except for the parameter T_i, which was chosen as $T_i = 0.0001$. The parameters for the periodic PI–controller were chosen as $K = 0.05$, $T_i = 3.0$, $T_a = 10.0$ except for the PFB, which was implemented with $K = 0.4$. The proportional gain of the PFB controller was chosen as $K_{pfb} = 0.003$. The feedback loop consisting of the PI controller and the linearized model of Eq. (4.46), including the external buffers, remains stable in a relevant range of operating points. Figure 7.15 shows the Bode diagrams for the operating points $\bar{w}$=5.04 ms, d_r=100 ms, M_c=128, and two different arrival rates; λ^0=50 req/s and λ^0=100 req/s. As seen in the figure, the stability margins remain high. The stability of the system with the feed–forward controllers have not been evaluated since only one of them (the QFF controller) is modeled.

The periodic PI–controllers used the comparison of the reference time and an average of the response times of the requests departed during the last sampling interval. Compared to the response time, the response time prediction already incorporates some averaging. Therefore, the response time and the response time prediction were pre–filtered separately with different filters, before being compared to form an estimation error for the predictor. All three filters were implemented as a first–order auto–regressive filters of the same structure as Eq. (7.12). The filter constant used for the response time and the response time prediction were $\alpha = 0.0005$ and $\alpha = 0.5$, respectively. The filter constant for the estimation

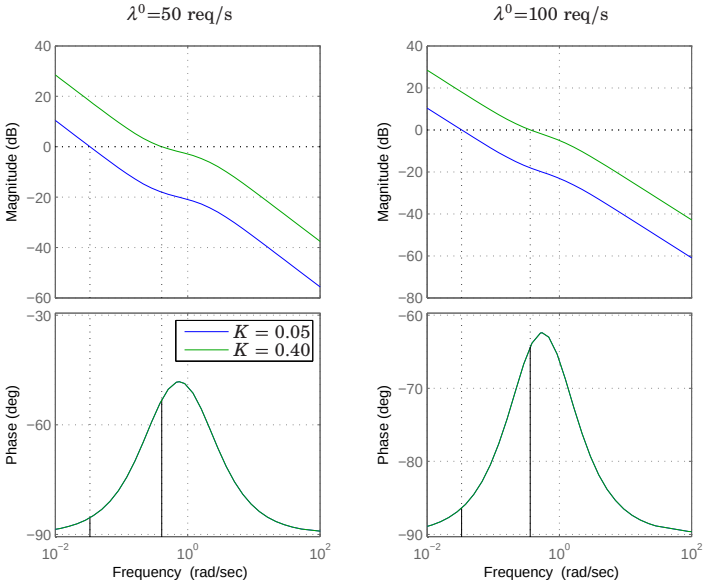

Figure 7.15 Bode diagrams of the open–loop transfer functions for the two controllers in the operation points λ^0=50 req/s and λ^0=100 req/s. The model includes the external buffers. The gain margins are infinite for all four cases, and the phase margins are between 93°and 127°. The black line indicate the phase margins.

error was $\alpha = 0.01$. The feed–forward signal from the IPFF controller was filtered with first–order auto–regressive filter with filter constant $\alpha = 0.5$.

Since the involved signals are very irregular, all time–domain results are presented as 30 s averages.

As in Section 7.2, the investigations are divided into steady–state investigations and transient investigations. The steady–state behavior illustrates how the controllers handle the short–term stochastic variations, while the transient investigations reveal the controllers' capability to handle larger changes in the work load. The latter is the main focus of the work presented in this thesis.

Steady–State Experiments

Fig. 7.16 shows results from a steady–state experiment with medium load (λ=50 req/s) after the transient period. Figure 7.17 also illustrates steady–state results in the time domain, but the accumulated variables give more direct information on how much the variables deviate from the desired values. The more steep a line is, the more the variable deviates

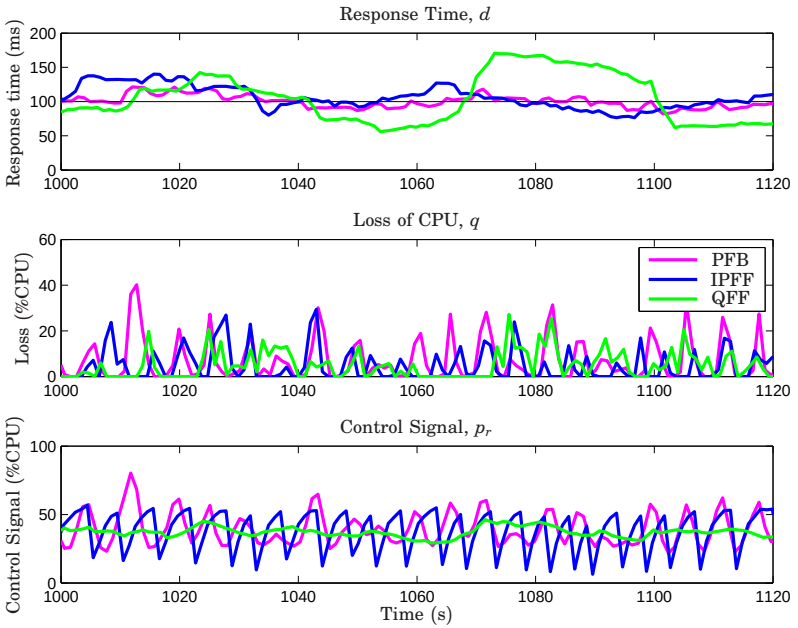

Figure 7.16 Experimental steady–state results in the time–domain. λ=50 req/s. $\hat{w}$=0.011 s, T_s=1 s, d_r=0.1 s.

from the desired value, so the desired line is horizontal, which cannot be obtained in practice. The figures reveal a trend similar for other work loads; The queuing–theory based controller (QFF) shows the worst capability to maintain a steady response time compared to the two other controllers. It is also observed that the control signal of the IPFF and the PFB controllers are more unsteady than that of the QFF controller.

Fig. 7.18 illustrates average results of a number of steady–state experiments of different average arrival rates λ. It shows that all the controllers are capable of maintaining the average response time near the reference. The IPFF and the PFB controllers showed similar capability to keep a steady response time average (similar variation costs) at least for medium and high load. The proposed PFB controller outperformed the other controllers with regards to the loss of CPU capacity at high load, but had the worst performance at low/medium load. These conclusions correspond well with the observations from the simulations.

The level of the loss of CPU capacities were all an order of magnitude higher than the simulation results with varying arrival rate (Fig. 7.7). This is expected to be due to the different response–time references; 1 s

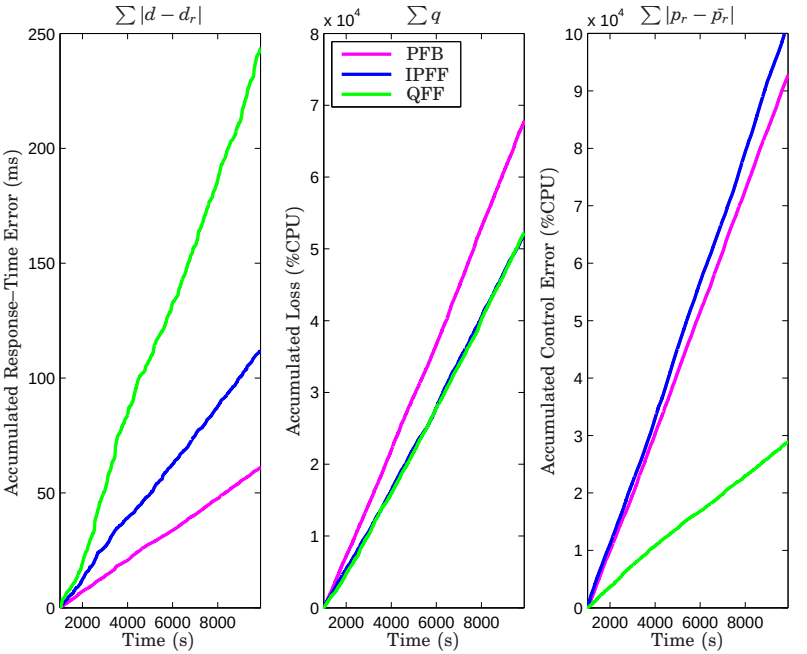

Figure 7.17 Experimental steady–state results. Accumulated response-error and accumulated loss of CPU capacity in the time–domain. λ=50 req/s. $\hat{w}$=0.011 s, T_s=1 s, d_r=0.1 s.

for the simulations and 0.1 s for the experiment. When the response times are smaller, fewer requests are being treated simultaneously, and there is a higher risk for the system to be empty occasionally and thus a higher loss of CPU capacity. The simulation results with varying response time reference presented in Fig. 7.9, support this argument.

Transient Experiments

The transient behavior of a controller shows how robust the controller is to changes in the system, for example changes in the arrival rate. Therefore, two sets of experiments with changing arrival rate were performed. In the first experiment, the system was initially exposed to a medium–load traffic with λ=50 req/s. After 150 s (at time t=0 s), the arrival rate doubled, so that the system was exposed to high–load traffic. In the second experiment, the system was initially exposed to high–load traffic, with λ=100 req/s. After 150 s (at t=0 s), the arrival rate decreased rapidly to λ=50 req/s. The disturbances are illustrated in Fig. 7.19.

Fig. 7.20 and Fig. 7.21 illustrate the results of the two sets of experi-

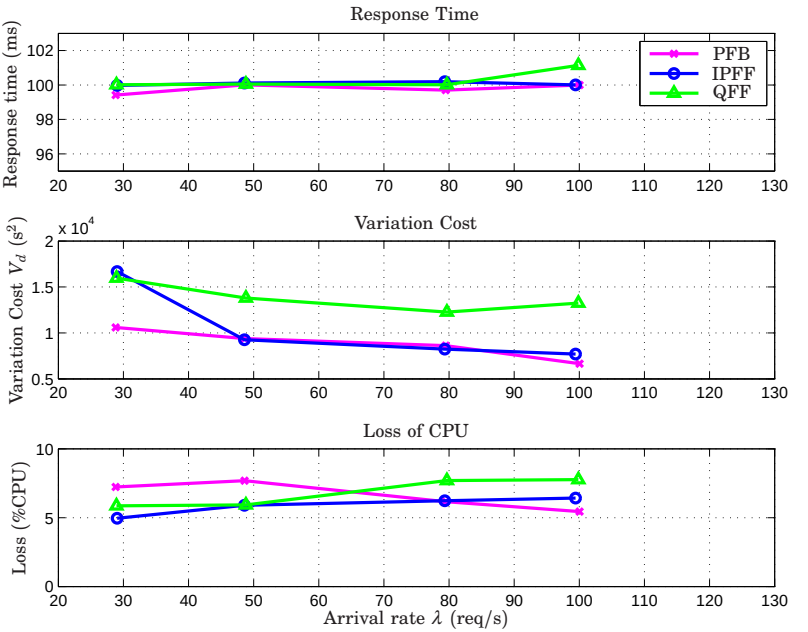

Figure 7.18 Averaged steady–state experiments for different arrival rates λ. $\hat{w}$=0.011 s, T_s=1 s, d_r=0.1 s.

ments. As can be seen in the figures, the queuing based predictor (QFF) performed rather badly. It over–reacted to the changes, thus spending too much CPU capacity. Also, it took quite a while to return to steady–state. The inverse–prediction controller (IPFF) did not react immediately to the change, but managed to recover relatively fast and with a relatively small increase in the response time. The feedback based prediction–controller (PFB) had the best performance. It reacted quickly to the change, and kept the increase in the response time to a minimum.

7.4 Discussions and Conclusions

In all transient evaluations, both by simulations and by experiments, the proposed PFB controller showed superior capability compared to the other controllers to suppress the effect of the change of work load. The improvement over the IPFF controller was not as pronounced as of the QFF controller, which can be related to the more advanced structure of both the PFB controller and the IPFF controller, which both incorporate measure-

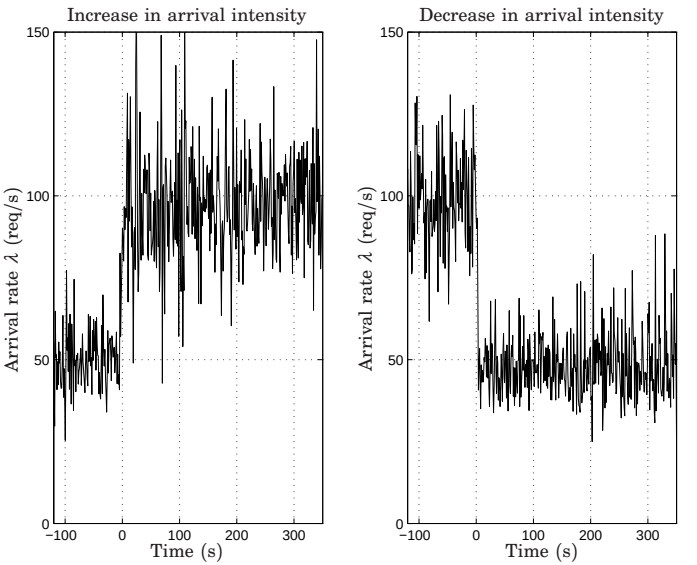

Figure 7.19 Disturbances (arrival rate) of the transient experiments.

ments of the number of jobs, $\tilde{n}$. In the experiments, the PFB controller was able to react faster to the change in arrival rate, and thus avoid a large deviation in the response time. Furthermore, it managed to return to a steady operation sooner.

A general trend in all the investigations was the poor performance of the queuing based controller (QFF). It is based on a fixed, offline estimated, required work, and only considers long–term averages in the feed–forward part. Only with low arrival rate, where the stochastics of the traffic became dominating, this controller performed similarly or a bit better than the other controllers. The transient behavior clearly indicates the problems of basing the feed–forward signal on offline estimations. In the presented results, the average required work used in the feed–forward controller were over–estimated. Since this estimate enters the feed–forward signal proportionally (see Eq. (7.8)), an over–estimate can have a dramatic effect as seen in all of the transient simulations and experiments; see Figs. 7.13 and 7.16. A solution is to reduce the estimate of the required work manually, but then the procedure is no longer systematic, and if it is lowered too much, the desired effect of the feed–forward mechanism diminishes. The simulations and the experiments clearly indicate that a control structure, where the required work is not estimated offline, is clearly preferable.

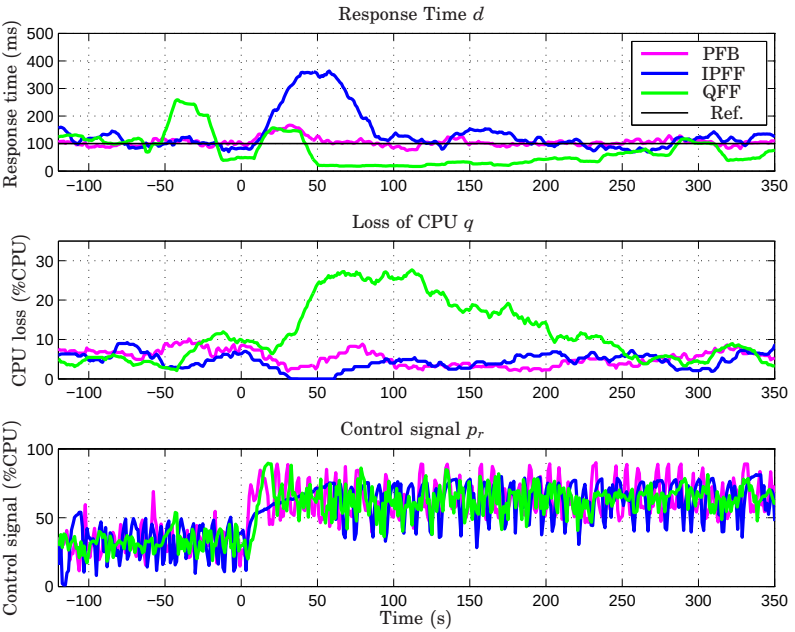

Figure 7.20 Time–domain transient experiment results with increasing arrival rate.

Further research is needed in order for the control scheme to be merged into more realistic setups. Especially the filtering issues need to be simplified. The proposed controller has a higher degree of complexity, where calculations are needed for each departure to update the predictor. Only the variables associated with the measurements are updated at the departures for the other two controllers, and the actual control calculations are only performed at sampling incidences. The predictor calculations for the proposed controller are of relatively low complexity, so an efficient implementation would yield a low overhead. The tuning of the predictor in the proposed controller requires more work by the operator than required for the two other controllers. Here further research is needed to establish sound tuning rules.

The most important difference between the proposed controller and the compared controller, is that the proposed controller does not require an offline estimation of the required work. This means that the controller has a superior transient behavior as it becomes robust to changes in the system. Also, in most of the scenarios, the proposed controller showed the best performance.

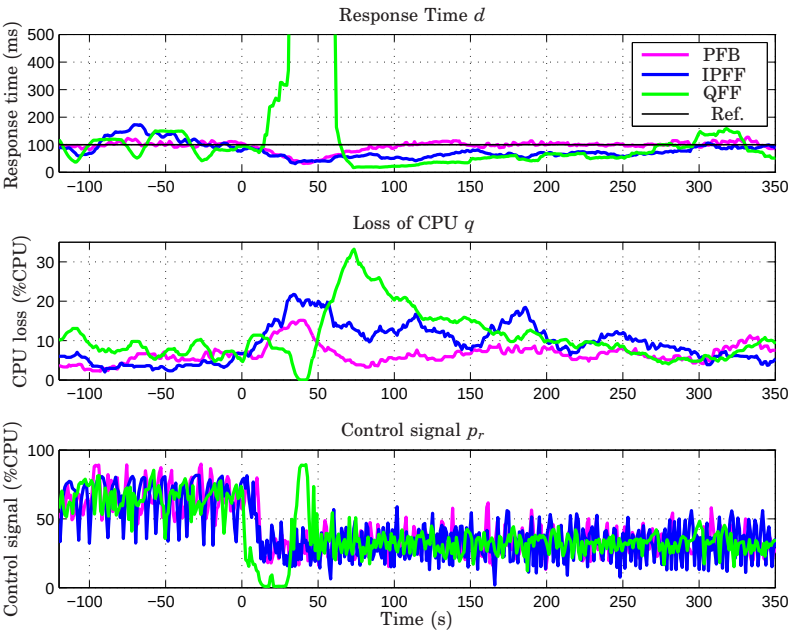

Figure 7.21 Time–domain transient experiment results with decreasing arrival rate.

8

Nice Resource–Reservation

The work presented in this chapter represents an early attempt of virtualization–like actuation for web–server control. The Linux kernel did not supply functionality to schedule the CPU capacity on group–level but only relative priority on process-level. A light–weight kernel–implementation based on existing operating–system infrastructure is presented along with tests on an experimental platform with Apache servers. The material presented in this chapter is based on [Ohlin and Kjær, 2007].

The objective of this chapter is to achieve CPU–capacity reservations on a standard Linux computer. Such reservations make it possible to reserve fractions of the CPU to specific tasks or groups of tasks (threads and processes).

The developed method has been implemented as an add–on to the Linux 2.6 kernel. Due to the changed scheduler policy from kernel 2.6.23 (released 2007) and onward the presented method is not compatible with newer Linux kernels.

A key factor in the implementation has been to make it non–intrusive and to preserve the way that the original scheduler works. This gives the benefit that the new features can be used without compromising existing functionality. The CPU–capacity controller uses the `nice` value as a control signal and the task's execution time as a measurement signal. This forces the scheduler to give the controlled tasks their specified amount of CPU capacity.

It may be argued that the presented problem can be solved offline by specifying a static nice value for each task. This is of course absolutely true if the system is static and everything is known in advance. That is, if it is known exactly how many tasks that are present in the system and also their execution–time demands. These premises are not likely to show up in an ordinary Linux desktop or server system and therefore it is necessary to introduce a feedback loop to cope with the unknown. In an ordinary computer system there is a lot of dynamics. This is due to the fact that tasks can arrive and leave the system at any time. Tasks can also change their state and consume more or less execution time. When running on multi–processor systems, tasks will also jump between processors in a more or less random pattern from a spectators' perspective. This causes the execution environment to change rapidly and therefore an ability to adapt to different situations is necessary.

8.1 Modeling

The scheduler of the Linux kernel is rather complicated, but some minor assumptions can lead to a static model suitable for control design.

There are three different scheduling policies available in Linux; SCHED_FIFO, SCHED_RR, and SCHED_OTHER. The two first are for soft real–time scheduling policies, and the last is for normal time–sharing scheduling. Only SCHED_OTHER is utilized in the work presented in this chapter. Every task gets to run a certain amount of time, denoted the time slice T_n , and the actual size of it depends solely on the nice value given to the task and not on the effective priority. nice values in the interval $[-20 \ldots 0 \ldots 19]$ are mapped to time–slice sizes in the interval [800 ms ... 100 ms ... 5 ms] as illustrated in Fig. 8.1. Note that the resulting time slices do not scale linearly with the nice value.

Model of the System

It is assumed that a task always needs to run, i.e., it is CPU–bound. The behavior of the scheduler can be summarized in a few assumptions and a model of the system to predict its behavior can be formulated.

- A task's time slice depends solely on its nice value.
- Tasks are scheduled in order of priority.

For a more detailed description of the task scheduler, see [Ohlin, 2006; Ohlin and Kjær, 2007].

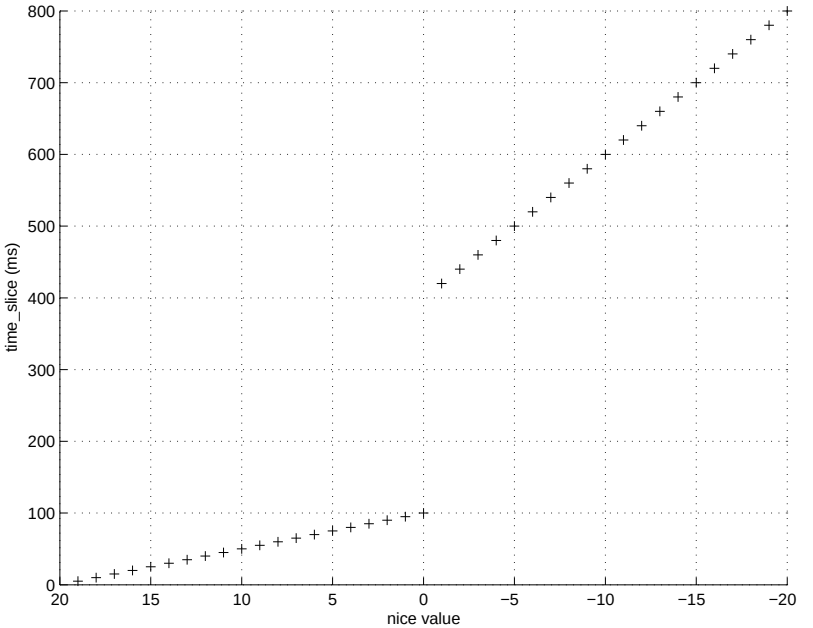

Figure 8.1 The size of `time slice` as a function of the `nice` value. Notice that the `nice` values are ordered from positive to negative values to indicate increasing `time slice` allocation.

According to the model, the fraction of the execution time a task is assigned during one round of execution, is calculated as:

$$p(j) = \frac{T_n(j)}{\sum_{\forall l} T_n(l)} \tag{8.1}$$

where j and l denote task indices.

Evaluation of the Model

To verify that the model is accurate, theoretical values from the proposed model are compared to values obtained through measurements. The experimental setup consists of four tasks running in endless while–loops. Three of the tasks have `nice` value 5 while the `nice` value of the fourth task is varied in order to give it more or less CPU capacity compared to the other tasks. The result can be seen in Fig. 8.2. Using lower `nice` values than −6 resulted in the system becoming too sluggish to do any good measurements. Therefore, these are left out. This sluggishness is probably due to the fact that tasks with that low `nice` values are given so high

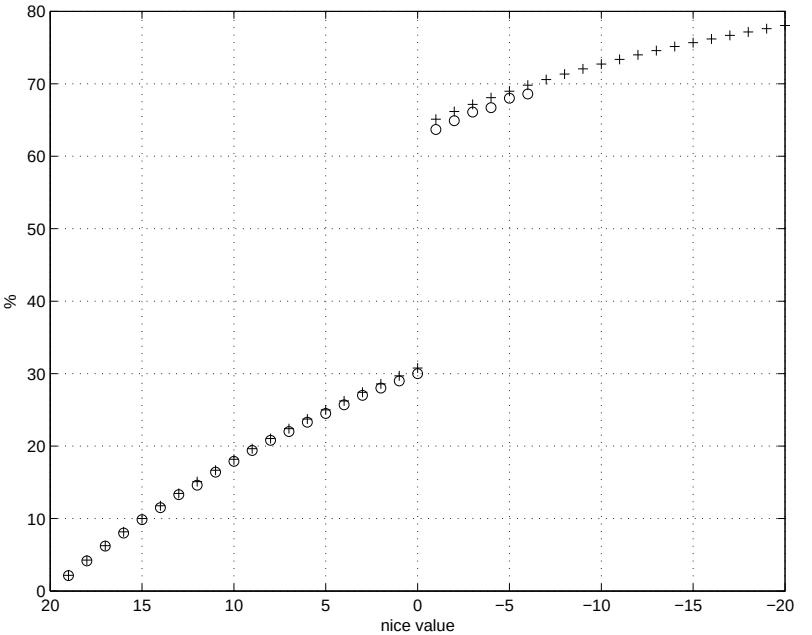

Figure 8.2 Comparison of the execution time model for Linux (+) and measurements on a computer (∘). Notice that the `nice` values are ordered from positive to negative values.

priorities that they conflict with the tasks interacting with the user. As can be seen, the results from the experiments follow the model very well.

8.2 Control Design and Implementation

The nonlinear mapping from the `nice` value to the `time_slice` is illustrated in Fig. 8.1. Special concern must be given to references, that requires the control signal to oscillate over the interval $[-1, 0]$. A PI controller with anti–windup has been implemented using fixed point arithmetic in a kernel module. Global knowledge of the behavior of other tasks could also be incorporated, but this has not been implemented in this work. By using a PI controller in this way, a modulation is achieved automatically. The implemented PI controller is given by

$$\begin{aligned} u(k) &= K\big(y_{ref} - y(k)\big) + I(k) \\ I(k) &= I(k-1) + T_s \frac{K}{T_i}\big(y_{ref} - y(k)\big) \end{aligned}$$

where the parameters K and T_i are the proportional and integral control parameters, respectively. The variable u is the control signal (the `nice` value), and y is the measurement (the fraction of time given to the task). The variable k is the discrete time index, so that $t = kT_s$, where T_s is the sampling time. The PI controller consists of two components. The proportional part ($K(y_{ref} - y(k))$) ensures fast reaction to disturbances, but does not assure that the desired reference is reached. The integral part (described by I) will accumulate any error between the measurement and reference in a similar manner as an incremental controller. This part is particularly beneficial when the system under consideration is not well–known and predictable as with computer systems. The specific implementation uses the control parameters $K = 0.0.01$ and $T_i = 52$ and sampling time of T_s=20 ms.

Since the control design is not based on a dynamical model, there are no theoretical guarantees for performance or stability. However, the values of K and T_i have been chosen rather conservatively to have large stability margins. This is imposed because even small changes to the `nice` value can lead to large changes in the achieved CPU–capacity. The effect of changes to the `nice` value also varies with the number of tasks in the system and their respective `nice` values. In the case where the controller saturates, the control objective may not be satisfied as the controller lacks actuation possibility. The integral part will remain within the allowed control range due to the anti–windup scheme.

For more details on the controller implementation, see [Ohlin, 2006].

Taking the Task's State Into Account

So far it has been assumed that a controlled task is always requesting CPU capacity, i.e., it is CPU bound. This may be true in some cases but obviously not in all. Imagine for example that the controlled task is given a reference of 50% but needs no more than 40% because it is occasionally waiting on some I/O. As the task does not require CPU capacity from time to time, the error will remain and the control signal will, due to the integral effect, continue to rise until it saturates. This is of course not a satisfying behavior, and could be avoided by taking the current state of the task into account when controlling it. The strategy could be; do not increase the control signal further if the task does not require CPU capacity. This is more or less an anti–windup scheme which ensures that the integral part does not wind up trying to enforce higher CPU allocation to a task than the task demands. Now the obvious question is: How to detect if a task is requiring more CPU capacity than it already has? The idea used in the current controller implementation is to sample the state of the task at the same time as the execution time. The controller is then only executed if two consecutive samples show that the task is in the

running state. This strategy works well if the task is mostly in the *running* state for a longer time than the time between two consecutive samples of the controller. How long time a task spends in its *running* state, depends highly on its workload during that time interval, but it also depends on the other tasks in the system as the task may get interrupted by a task of higher priority. This makes it hard to give any general rules and hence draw any conclusions to be used for more accurate control.

8.3 Verification by Experiments

The reservation method was tested under different conditions. First, the method was tested on a standard PC and a number of while loops to consume CPU capacity. Then a more realistic scenario was used, involving web servers and stochastic traffic from several clients.

Experiments with Load Tasks

Two experiments have been performed on a desktop computer with a single CPU. At the same time as the experiments were made, there were a number of tasks in the system, e.g., *X*, *Firefox*, *Thunderbird*, *XEmacs*, and so on. All CPU–capacity measurements have been filtered through a moving average window of 4 s. The filtering is done because of the fact that when a task executes, it gets 100% of the CPU capacity and then it gets 0% when it does not execute. Filtering through a moving average window shows the CPU capacity during that window.

Running the experiments on a computer with more than one CPU will gain results similar to the ones seen in this section, except that there will be considerably more load disturbances.

The first experiment consists of four tasks running in endless while–loops. Two of the tasks have their `nice` values set to 5, and act as background load. The third and fourth task's `nice` values are used as control signals to keep the measured CPU–capacity at the desired references.

The references for both of the tasks are initially kept at 25%. At time 182 s, the reference for the first task is changed from 25% to 50%. At around time 320 s, the reference is changed back to 25%. The result of the step response for the first task can be observed in Fig. 8.3. The coupling between the two tasks is visible in Fig. 8.4, which shows the disturbance on the second task as a consequence of the step on the first one. No feed–forward term is used in the controller. This experiment shows the modulating nature of the control signal and the results of the quantization in the `nice` value. In Fig. 8.3, it can be seen that the control signal is constant both before and after the two steps. But when the reference is

set to 50%, the control signal fluctuates a lot. Also note that there is much less oscillation in the CPU capacity when the reference is set to 25% than when it is set to 50%. This is due to the fact that some references cannot be kept stationary because the `nice` value is discrete.

The second experiment consists of one periodic task that executes for approximately 40 ms and then sleeps for 60 ms repeatedly. This results in a task that uses 40% of the CPU at most even if it is alone in the system. Controlling such a task requires the state of the task to be taken into account as described in Section 8.2. Two load tasks of the same type used in the previous experiment are also present in the system.

As can be seen in Fig. 8.5, the proposed scheme works well in practice. In the beginning of the plot, the reference is higher than the task demands, and at time 55 s it is set to an even higher value but the control signal still behaves well. It is also observed that the controller is still able to follow reference changes when they are lower than 40%.

The observant reader may notice the delay and the following under–shoot at time 160 s. Also note that this behavior does not show up at any of the other step changes in the plot. This is not an integrator windup as may be assumed first, but is instead due to the fact that the system has marked the task as interactive and therefore given it an additional bonus. When the task after some time is marked as non–interactive, it loses its bonus and this causes the under–shoot.

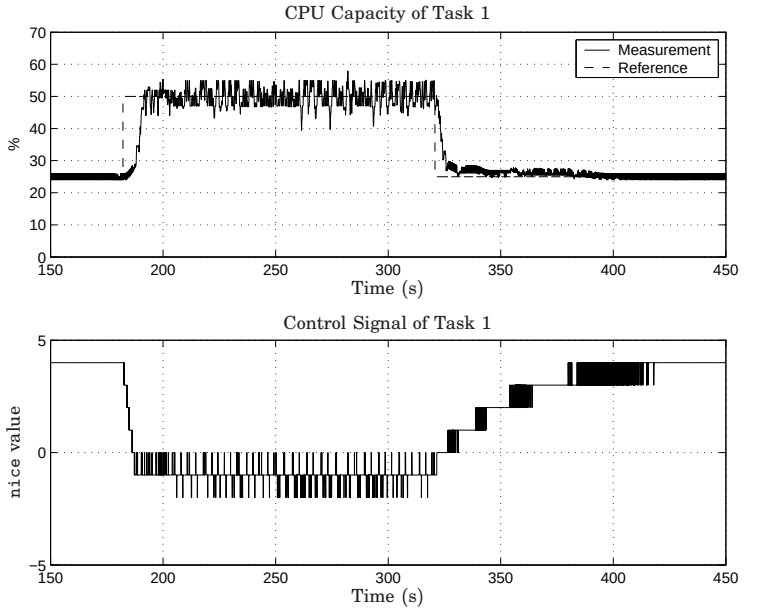

Figure 8.3 Step response of the CPU capacity (task 1) when controlling two tasks.

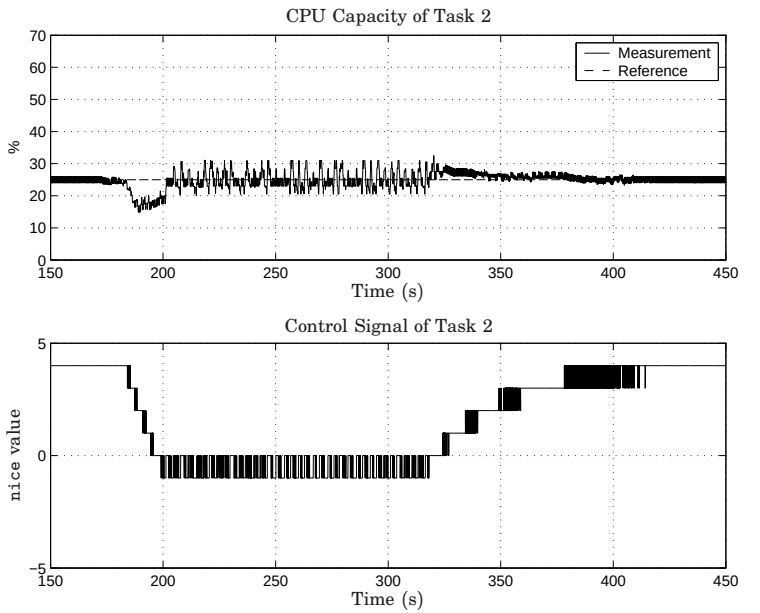

Figure 8.4 Step response of the CPU capacity (task 2) when controlling two tasks.

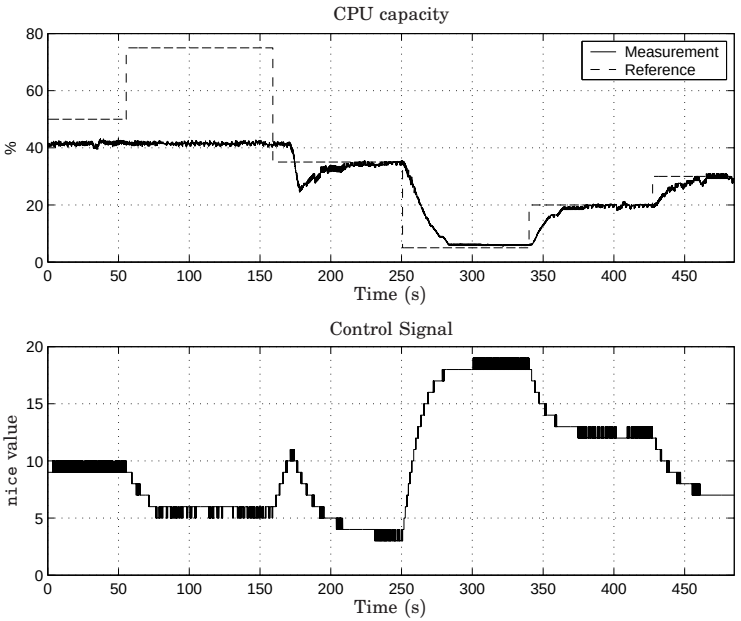

Figure 8.5 Step responses for a non CPU–bound task when taking the task's state into account.

Experiment with Apache Servers

To test the scheduling mechanism on a more realistic application, experiments were conducted on a testbed with an Apache server. As described earlier in this thesis, the Apache server creates processes as they are required, and destroys them again when they are no longer needed. This non–constant amount of processes, along with the stochastics of the requests, resulted in non–smooth CPU–capacity requirement, which served to stress the reservation method. The test is not included to suggest that the mechanism is the best solution for the given example, but only to demonstrate that the method works for a problem including more advanced behavior than the previous tests.

The setup represents a hosting system where two service providers are hosted on the same physical hardware. The objective is to separate the two service providers, so that one service provider can operate unaffected by a request overload at the other service provider.

The physical hardware used for this experiment is the same as presented in Chapter 3, but the configuration of the server is different. Fedora 5 (kernel 2.6.17) was used as operating system for the servers. Two Apache servers (version 2.2.2), configured by using the `prefork` module, were installed on the server as two distinct service providers. The controller was configured to set a common `nice` value to all the processes of a given Apache server. Also, the CPU capacity allocated to all the processes of one Apache serve were summed to give the time fraction measurement. In this manner, a single–input/single–output (SISO) system was obtained as required by the control structure. Only one of the Apache servers was controlled with the proposed scheduling mechanism. The controlled and uncontrolled server were listening on port 80 and 81, respectively. The setup is illustrated in Fig. 8.6.

Traffic was generated from the 12 client computers described in Chapter 3. The clients were grouped into three equal groups, each consisting of four client–computers. The traffic was generated using the traffic generation software CRIS [Hagsten and Neis, 2006]. All clients requested the same PHP file, generating a response with 7000 characters, with exponential distributed inter–arrival times. The clients were configured to timeout after 10 s.

At the beginning of the experiments, client group I started sending requests to the controlled server, and client group II started to sent requests to the uncontrolled server. Both groups sent approximately 160 req/q. This traffic did not result in CPU overload, but left approximately 25% of the CPU capacity free. A server is considered to be overloaded when the requests cannot be served within the timeout of the clients due to lack of CPU resources. After 173 s, client group III started to send approximately

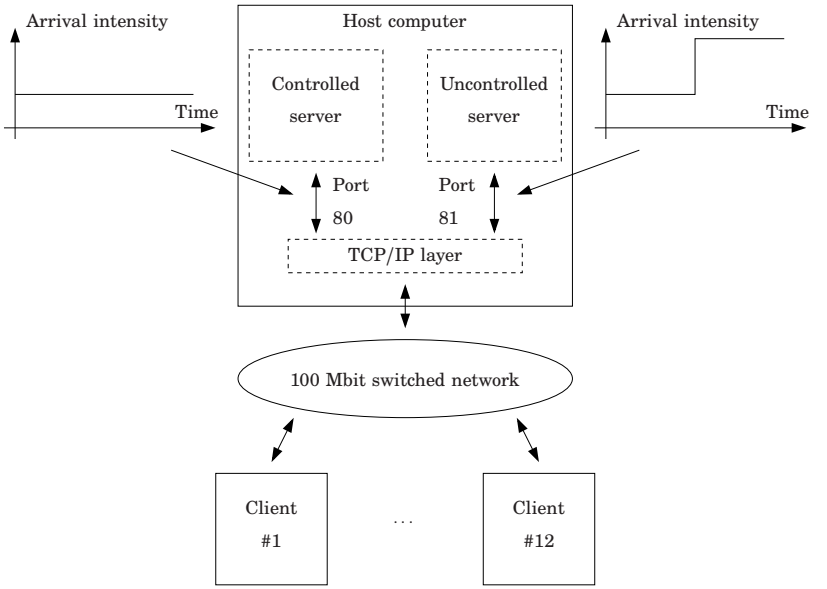

Figure 8.6 Experimental setup for experiment with Apache servers.

160 req/q to the uncontrolled server. The traffic going to the uncontrolled server was the combined traffic from client group II and III. The server did not have sufficient CPU capacity to maintain operation of both servers. The average arrival–rate is shown in the top of Fig. 8.7. Preferably, only the server being exposed to the extra traffic should become overloaded while the other server should remain operational.

The middle and bottom of Fig. 8.7 illustrate the response times of the controlled server and the uncontrolled server, respectively. In the case where the controller was inactive, both servers became overloaded when the traffic increased. After the increase of traffic, the response times of both servers increased dramatically and all clients started to timeout. Consistent timeouts from the clients were observed. In the case where the CPU–capacity allocation was controlled by the proposed scheduling mechanism (reference set to 45% of the CPU capacity), only the uncontrolled server became overloaded. The controlled server continued to perform with similar response times. Client timeouts were observed consistently only on the uncontrolled server's requests. Two single timeouts were observed on the controlled server's requests. The small jump observed in the response time of the controlled server (the middle of Fig. 8.7) at around 200 s is assumed to be due to some disturbance from other tasks in the operating system.

This experiment showed that the scheduling mechanism can be used to affect the response time of a web server application. The setup with two different servers with different client groups, can be used in applications where different sites are hosted on the same physical computer, but where the performance of one server must be independent of the behavior of the other server. In this experiment, it has not been considered how such a system should be built in a real application. However, the experiment showed that the two servers can be separated by means of the proposed scheduling mechanism. The setup does not aim to control the response time. If this was the objective, a second control loop would have to be included, defining the reference for the CPU–capacity controller.

8.4 Discussions and Conclusions

This chapter has presented an attempt to supply functionality to reserve CPU capacity in a manner not offered by the original operating system. It can be viewed as a CPU virtualization method, however, it does not provide any guarantee that the requested reservation is actually met, especially not during load changes or reference changes.

An extension of the Linux 2.6 scheduler has been proposed, and a feedback based method for controlling the CPU capacity given to tasks in Linux, has been presented. The presented method has shown to work, both for CPU bound and non CPU bound tasks. A number of experiments have been performed in order to show that the technique works in reality. The experiments indicate that CPU–capacity allocation can be obtained with the proposed scheduling mechanism. Furthermore, the mechanism can be used to separate the performance of two Apache servers running on the same physical computer.

It should be mentioned that the method presented here is not compatible with newer Linux kernels as the scheduler has been changed significantly from kernel 2.6.23. The reservation method described in this chapter has dynamics in the time scale of seconds, while the scheduler of the new Linux kernel has dynamics in the time scale of a tenth of a second (see the discussion on sampling interval in Section 3.7 on page 54).

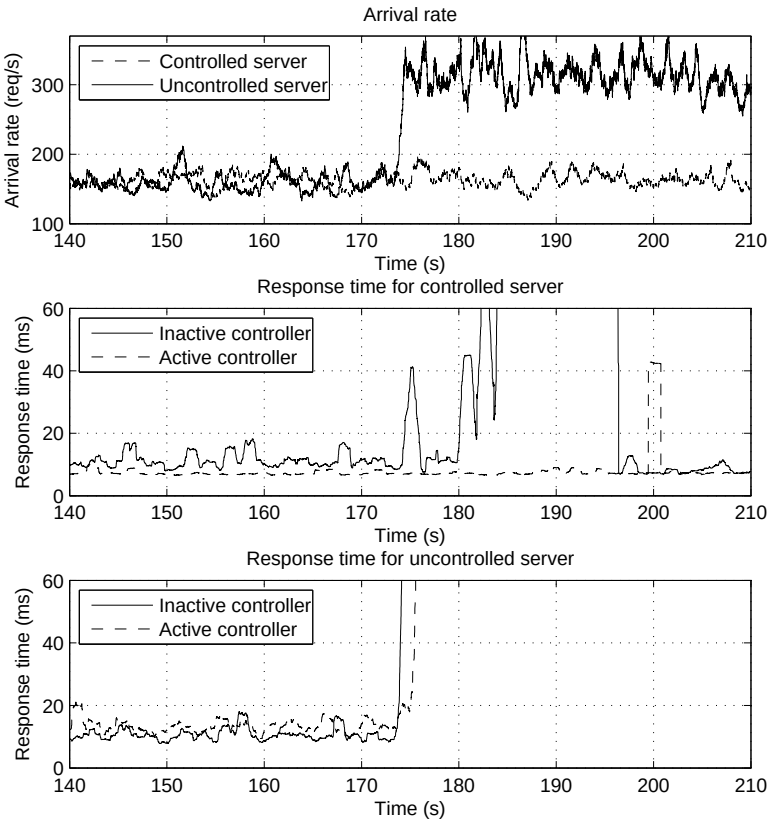

Figure 8.7 Results from experiment on two Apache servers. Top: Average arrival rates. Middle: Response time for the controlled server with and without feedback control. Bottom: Response time for the uncontrolled server. All variables were measured at the clients and were filtered with a moving average window of 200 requests.

9

Concluding Remarks and Future Work

9.1 Concluding Remarks

The focus of this thesis has been response–time control of web–server systems. The web–server system providers and academia have shown an increasing interest in this field in order to supply reliable services to costumers while maintaining a low operation–cost. Over the last years, the environmental impact of computing has also come into consideration, something that encourages to optimize the performance even further.

In order to optimize the resource consumptions, more advanced control algorithms must be investigated, and the field of automatic control has much to offer. It is well known that a possible hazard of tuning for fast and accurate control is instability, which has devastating consequences for the performance. Therefore, modeling and analysis of the behavior of web–server systems have become more and more important. Results presented in this thesis explain the origin of some instability observed on a web–server system setup. It is concluded, and verified by both simulations and experiments, that this instability arises if the controller reacts too aggressively to a change in the arrival rate. This clearly demonstrates the danger of focusing on the performance. The same analysis also revealed that the root of the instability was inaccurate modeling. It was assumed that measurements could be performed inside the web–server system without considering buffering earlier in the software layers. The analysis showed that these buffers introduce some important dynamic phenomena, which change the behavior of the controlled system completely.

In the pursuit of a resource–optimized web–server, this thesis presents an improved feed–forward mechanism to compensate for changes in ar-

rival rate at the web server. A previously proposed method was based on an offline estimated variable, which might change over time during operation. The method, suggested in this thesis, adjusts this parameter online, and thereby ensures that the feed–forward mechanism adapts to the changing needs of the clients. The new feed–forward method uses a model to predict the future response times and relocates resources in order to compensate for the changes before they propagate to the response times. The method was validated by both simulations and experiments, and proved to be superior to other methods from the literature.

Finally, this thesis presented a solution to obtain CPU–capacity reservation in the Linux kernel. The method allows the user to define how much of the CPU capacity (in percentage) a process, or a group of processes, are guaranteed. The method uses only existing kernel–functionality modulation of the so called `nice` value, which is the operating–system priority method. The reservation scheme is tested on a testbed with two Apache servers to demonstrate the performance in a situation where the requested CPU–capacity changes much over time.

Two important assumptions have been made. A perfect balancing mechanism is working in the computer system, and there are always sufficient resources to respect the demands from all applications. In particular, the latter is restrictive. When the total load is higher than the available resources, choices have to be made on how to distribute the available resources. The applications can have quite different requirements and demands, and the applications can be associated by multiple computers. The allocation problem can render a complicated optimization problem, especially if also the dynamics of the system are taken into account.

9.2 Future Work

The work presented in this thesis related to response–time control should be generalized to multi–tier systems. Ideas on this subject have already been presented by others, see *e.g.* [Jayachandran *et al.*, 2009; Padala *et al.*, 2009a; Padala *et al.*, 2009b].

The Apache server has a simple resource controller, as described in Section 2.3. Here, the amount of CPU capacity obtained by the server is given by the number of jobs present in the system, assuming that the amount of other processes remains constant. This could be modeled by assuming that the CPU resources are divided equally between all active processes, and thus

$$p_r(t) = \frac{n(t)}{n(t) + n_0}$$

where n_0 is the (constant) number of processes other than those of the server. Combined with the model presented in Chapter 4, the stability of the resource controller can be evaluated. This is a rather academic problem as the Apache server has been running stable on thousands of real sites. However, other resource controllers could be investigated, such as the controllers which use the maximum number of requests as actuator [Hellerstein *et al.*, 2005; Lindegren, 2008].

Interesting results on the subject of content–adaptation were presented in [Andersson, 2007]. Here, the network bandwidth was considered to be a bottleneck during high load situations, and the file size of static files were divided into five different levels. It was assumed that all the file sizes were known along with the maximum bandwidth, the population distribution, and the arrival rate. Then the content level of the individual files were optimized according to a defined cost–function. The optimization was performed offline for the specific operating point and utilized in steady–state experiments. From a control–theoretic point of view, this method has several problems:

1. The specific implementation could not handle changes in the environment (such as changes in the popularity distribution or in the arrival rate) because the optimization was performed offline.

2. Even if the optimization was performed online, the control strategy is purely feed–forward, as only un–affectable inputs are included in the optimization (the popularity distribution and the arrival rate). If the model is not accurate, the desired result will never be met as the controller will not be aware of the effects of its own behavior.

The first point is rather straight–forward to solve, assuming that the optimization algorithm can be executed in a sufficiently efficient way. Alternatively, an offline optimization over the full input–space could be implemented as a look–up table in the memory. This would allow changes in the inputs and thereby investigation of the dynamics of the controller.

Inclusion of feedback into the controller would potentially solve the second problem, but this is not trivial. The optimization used does not directly allow for any feedback, so the optimization will probably have to be redesigned. Tools from *Model Predictive Control* (MPC) could hold solutions to the problem.

A

Nomenclature

Latin uppercase

A	Amplitude in describing function
C^2	Squared variance coefficient
$D(s)$	Laplace transform of d
$D_i(s)$	Laplace transform of d_i
$D_r(s)$	Laplace transform of d_r
$G(s)$	Transfer function
$G_0(s)$	Open–loop transfer function
$G_c(s)$	Desired compensation link
$G_{cl}(s)$	Closed–loop transfer function
$G_{fb}(s)$	Compensation link for the feedback controller
$G_{ff}(s)$	Compensation link for the feed–forward controller
$G_s(s)$	transfer function for ideal band–stop filter
H	Ensemble average utilization
H_2	Second order hyper-exponential distribution
I	Integral part of the PI controller
J_d	Response time cost–function
J_q	CPUcapacity cost–function
K	Proportional coefficient of the PI controller
K_{pfb}	Feedback coefficient
K_{ss}	Steady–state gain of first–order linear system
M_c	Maximum number of requests in the server
$N(s)$	Laplace transform of n

$N_i(s)$	Laplace transform of n_i
P	Filtered signal
$P_{fb}(s)$	Laplace transform of p_{fb}
$P_{ff}(s)$	Laplace transform of p_{ff}
$P_r(s)$	Laplace transform of p_r
Q	Filtered signal
S	Closed curve in the complex plane
T_a	Anti windup time–constant of the PI controller
T_i	Integral coefficient of the PI controller
T_l	Time constant of first–order linear system
T_n	Time slice, the time period a task is given by the scheduler
T_s	Constant sampling interval
T_d	Time constant of first–order linear filter
T_λ	Time constant of first–order linear filter
$U(s)$	Input of $G(s)$
$\hat{U}(s)$	Compensated input
V_d	Variation
$Y(s)$	Output of $G(s)$

Latin lowercase

a	Arrival times of requests
a_1	Parameter in the describing function method
b_1	Parameter in the describing function method
c_0	Parameter for pole placement
c_1	Parameter for pole placement
d	Response time
$\bar{d}$	Average response time
$\hat{d}$	Estimated response time
d_i	Measured response time
d^f	Filtered response time
d^p	Filtered response time
d_r	Response time reference
d^w	Average response time of the jobs that departed during the interval between sampling $k-1$ and k

$f(\cdot)$	Static nonlinear function
$f_i(\cdot)$	Static nonlinear function
$\tilde{f}_i(\cdot)$	Static nonlinear function
$g_{\lambda,n}$	Steady–state gain from λ to n.
$g_{\mu,n}$	Steady–state gain from μ to n.
h	Sampling interval
i	Imaginary operator ($\sqrt{-1}$)
j	Summation index
k	Discrete–time index
l	Summation index
m	Integer number
n	Expected number of jobs in the system as a dynamical state
$\tilde{n}$	Instantaneous number of jobs/requests in the system
$\bar{n}$	Average number of jobs/requests in the system
n_0	Processes other than those of the server
n_e	Number of jobs in the external queues
$\bar{n}_e$	Average number of jobs/requests in the externals queues
$\bar{n}^f$	Filtered number of jobs
n_i	Measured number of jobs in the system
$\bar{n}_i$	Average number of jobs/requests in the internal system
p	Fraction of the CPU used by the server
p_a	Fraction of the CPU used by the server
p_{fb}	Feedback control signal
p_{ff}	Feed–forward control signal
p_r^i	Fraction of the CPU reserved to the i–group. The notation p_r is equivalent to p_r^{server}
p_{ipff}	Feed–forward signal from the IPFF controller
p_j	Probability for the queuing system to hold j jobs
p_p	Predictive control signal
p_{qff}	Feed–forward signal from the QFF controller
q	Loss of CPU resources, $q=p_r$-p_a

s	Laplace operator
t	Time
t_{now}	Time at the departure of a particular request
t_t	Transient period
u	General process–input
v	Non–saturated control signal
w	Required work
$\bar{w}$	Average required work
$\hat{w}$	Estimated average required work
x	Service time
$\bar{x}$	Average service time
y	General process–output
y_{ref}	General process–reference
z	Virtual state

Greek uppercase

$\Lambda(s)$	Laplace transform of λ
$\Lambda_f(s)$	Laplace transform of λ_f
$\Lambda_i(s)$	Laplace transform of Λ
$\Psi(A)$	Describing function

Greek lowercase

α	Filter constant
β	Hyper-exponential distribution constant
γ_1	Coefficient in server model
γ_2	Coefficient in server model
γ_n	Coefficient in server model
γ_λ	Coefficient in server model
$\Delta\psi$	Linearization variable of ψ
ζ	Damping factor
λ	Arrival rate (requests/s)
$\hat{\lambda}$	Estimated average arrival rate (requests/s)
λ_f	Filtered average arrival rate (requests/s)

λ_i	Measured average arrival rate (requests/s)
μ	Service rate (requests/s)
ξ_1	Variable
ξ_2	Variable
ρ	Load
σ_f	Coefficient in server model
σ_n	Coefficient in server model
σ_λ	Coefficient in server model
τ	Integration variable
ψ	A variable or sequence
ω	Frequency
ω_c	Frequency
$\hat{\omega}_c$	Resonance–frequency estimate
ω_s	Stop–frequency of ideal band–stop filter

Abbreviations

CERN	European Organization for Nuclear Research
CGI	Common Gateway Interface, a scripting language for producing dynamic web pages
CPU	Central Processing Unit
CRIS	Crisis Request generator for Internet Servers
DVS	Dynamic Voltage Scaling
FCFS	First Come, First Served
ftp	file transfer protocol
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IIS	Internet Information Server
IP	Internet Protocol
IPFF	Inverse Prediction Feed–Forward
MPC	Model Predictive Control
OSI	Open Systems Interconnection
PHP	Hypertext PreProcessor. A scripting language for producing dynamic web pages
PFB	Predictive FeedBack

PS	Processor Sharing
QFF	Queuing–theory based Feed–Forward
QoS	Quality of Service
RED	Random Early Detection
RUBiS	Rice University Bidding System
SISO	Single input, single output
SLA	Service Level Agreement
ssh	secure shell
SURGE	Scalable URl Reference GEnerator
TCP	Transmission Control Protocol
URL	Uniform Resource Locator

B

Bibliography

Abdelzaher, T. and N. Bhatti (1999): "Web content adaptation to improve server overload behavior." *Computer Networks*, **11–16**, pp. 1563–1577.

Agnew, C. (1976): "Dynamic modeling and control of congestion-prone systems." *Operations Research*, **24:3**, pp. 400–419.

Anderson, D., D. Sweeney, and T. Williams (1998): *Statistics for Business and Economics*, seventh edition. South–Western College Publishing.

Andersson, M. (2007): *Overload Control and Performance Evaluation of Web Servers*. PhD thesis, Dep. of Communication Systems, Lund University, Sweden.

Apache Software Foundation (2008): "Developer documentation for apache 2.0." http://httpd.apache.org/docs/ 2.0/developer/.

Årzén, K.-E. (1999): "A simple event-based PID controller." In *Proc. 14th World Congress of IFAC*, pp. 423–428. Beijing, P.R. China.

Åström, K. (2006): *Introduction to Stochastic Control Theory*. Dower publications Inc, Mineola, NY.

Åström, K. and T. Hägglund (2005): *Advanced PID Control*. ISA–The Instrumentation, Systems, and Automation Society, Research Triangle Park, NC.

Åström, K. and B. Wittenmark (1997): *Computer–Controlled Systems*. Printice Hall, Upper Saddle River, NJ.

Åström, K. J. and T. Hägglund (1984): "Automatic tuning of simple regulators with specifications on phase and amplitude margins." *Automatica*, **20**, pp. 645–651.

Åström, K. J. and R. M. Murray (2008): *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, Princeton, NJ.

Barford, P. and M. Crovella (1998): "Generating representative web workloads for network and server performance evaluation." In *Proc. Performance '98/ACM SIGMETRICS '98*, pp. 151–160. Madison WI.

CKRM (2009): "Class-based kernel resource management." http://ckrm.sourceforge.net/.

de Jongh, J. (2002): *Share Scheduling in Distributed Systems*. PhD thesis, Delft University of Technology.

Elnozahy, E., M. Kistler, and R. Rajamony (2003): "Energy-efficient server clusters." In *Lecture Notes in Computer Science 2325*, pp. 179–197. Springer-Verlag Berlin Heidelberg.

Epema, D. H. J. (1998): "Decay-Usage Scheduling in Multiprocessors." *ACM Tran. Computing System*, **16:4**, pp. 367–415.

Erlang, A. (1909): "Sandsynlighedsregning og telefonsamtaler." *Nyt Tidsskrift for Matematik*, **B:21**, p. 33.

Essick, R. B. (1990): "An Event-Based Fair Share Scheduler." In *Proc. of the Winter 1990 USENIX Conf.*, pp. 147–162. USENIX.

Fong, L. L. and M. S. Squillante (1995): "Time-Function Scheduling: A General Approach to Controllable Resource Management." Technical Report RC 20155 (89194). IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY.

Franklin, G. F., J. D. Powell, and A. Emami-Naeini (1994): *Feedback Control of Dynamic Systems*. Addison–Wesley.

Fu, Y., H. Wang, C. Lu, and R. Chandra (2006): "Distributed utilization control for real-time clusters with load balancing." In *Proc. 27th IEEE Int. Real-Time Systems Symposium (RTSS'06)*, pp. 137–146. IEEE, Rio de Janeiro, Brazil.

Hagsten, A. and F. Neis (2006): "Crisis request generator for internet servers.". Master's thesis, LTH, Lund University.

Hellerstein, J. L. (1993): "Achieving Service Rate Objectives with Decay Usage Scheduling." *IEEE Trans. Software Eng.*, **19:8**, pp. 813–825.

Hellerstein, J. L. (2004): "Challenges in Control Engineering of Computing Systems." In *Proc. of the 2004 American Control Conf.*, vol. 3, pp. 1970–1979.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2004): *Feedback Control of Computing Systems*. Wiley-Interscience.

Hellerstein, J. L., Y. Diao, S. Parekh, and D. M. Tilbury (2005): "Control Engineering for Computing Systems." *IEEE Control Systems Magazine*, **25:6**, pp. 56–68.

Henriksson, D., Y. Lu, and T. Abdelzaher (2004): "Improved prediction for web server delay control." In *Proc. 16th Euromicro Conf. on Real–Time Systems (ECRTS'04)*. Catania, Italy.

Henry, G. J. (1984): "The Fair Share Scheduler." *AT&T Bell Laboratories Technical Journal*, **63:8**, pp. 1845–1857.

Heo, J., D. Henriksson, X. Liu, and T. Abdelzaher (2007): "Integrating adaptive components: An emerging challenge in performance–adaptive systems and a server farm case–study." In *Proc. 28th International Real-Time Systems Symposium (RTSS 2007)*, pp. 227–238. IEEE.

Horvath, T., T. Abdelzaher, K. Skadron, and X. Liu (2007): "Dynamic voltage scaling in multitier web servers with end-to-end delay control." *IEEE Transactions on Computers*, **56:4**, pp. 444–458.

Jacobson, V. (1988): "Congestion avoidance and control." In *SIGCOMM*, pp. 314–329.

Jayachandran, J. H. P., I. Shiny, D. Wang, and T. Abdelzaher (2009): "OptiTuner: An automatic distributed performance optimization service and a server farm application." In *Proc. Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'09)*. San Francisco, CA.

Kay, J. and P. Lauder (1988): "A fair share scheduler." *Communications of the ACM*, **31:1**, pp. 44–55.

Khalil, H. K. (2002): *Nonlinear Systems*, third edition. Prentice Hall, Upper Saddle River, NJ.

Kihl, M. (1999): *Overload Control Strategies for Distributed Communication Networks*. PhD thesis, Dep. of Communication Systems, Lund University, Sweden.

Kihl, M., A. Robertsson, M. Andersson, and B. Wittenmark (2007): "Control-theoretic analysis of admission control mechanisms for web server systems." *The World Wide Web Journal, Springer*, **11:1**, pp. 93–116. Online Aug 2007, print March 2008.

Kjær, M. A. (2005): "Active stabilization of thermoacoustic oscillation." Licentiate Thesis ISRN LUTFD2/TFRT--3239--SE. Department of Automatic Control, Lund University, Sweden.

Kjær, M. A., R. Johansson, and A. Robertsson (2006): "Active control of thermoacoustic oscillation." In *Proceedings of the IEEE International Conference on Control Applications*, pp. 2480–2485. Munich, Germany.

Kjær, M. A., M. Kihl, and A. Robertsson (2007): "Response-time control of a single server queue." In *Proc. 46th IEEE Conference on Decision and Control (CDC'07)*, pp. 3812–3817. New Orleans, LA.

Kjær, M. A., M. Kihl, and A. Robertsson (2008): "Response-time control of a processor-sharing system using virtualized server environments." In *Proc. 17th IFAC World Congress*, pp. 3612–3618. Seoul, Korea.

Kjær, M. A., M. Kihl, and A. Robertsson (2009): "Resource allocation and disturbance rejection in web servers using SLAs and virtualized servers." *IEEE Trans. Network and Service Management*. Submitted.

Kjær, M. A. and A. Robertsson (2009): "Effects of neglecting buffers in feed–forward design for web servers." In *Proc. Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'09)*, pp. 61–68. San Francisco, CA.

Kjær, M. A. and A. Robertsson (2010): "Analysis of buffer delay in web–server control." In *Proc. American Control Conference (ACC'10)*. IEEE, Baltimore, Maryland. Submitted.

Kleinrock, L. (1967): "Time–shared systems: A theoretical treatment." *Association for Computing Machinery*, **14:2**, pp. 242–261.

Kleinrock, L. (1975): *Queuing Systems*. John Wiley & Sons, Inc, New York.

Kremien, O. and J. Kramer (1992): "Methodical analysis of adaptive load sharing algrorithms." *IEEE Trans. on Parallell and Distributed Systems*, **3:6**, pp. 747–760.

Kuri, J. and A. Kumar (1995): "Optimal control of arrivals to queues with delayed queue length information." *IEEE Trans. Automatic Control*, **40:8**, pp. 1444–1450.

Laurie, B. and P. Laurie (2002): *Apache: The Definitive Guide*, third edition. O'Reilly.

Lee, S., J. Lui, and D. Yau (2004): "A proportional–delay diffserv–enabled web server: Admission control and dynamic adaptation." *IEEE Tran. Parallel and Distributed Systems*, **15:5**, pp. 385–400.

Lindegren, E. (2008): "Preparing the Apache HTTP server for feedback control application." Master's Thesis ISRN LUTFD2/TFRT--5796--SE. Department of Automatic Control, Lund University, Sweden.

Linux Headquarters (2008a): "This is the cfs scheduler." http://www.linuxhq.com/kernel/v2.6/25/Documentation/sched-design-CFS.txt.

Linux Headquarters (2008b): "What are cgroups ?" http://www.linuxhq.com/kernel/v2.6/25/Documentation/cgroups.txt.

Linux Headquarters (2009): "Memory resource controller." http://www.linuxhq.com/kernel/v2.6/25/Documentation/controllers/memory.txt.

Liu, X., J. Heo, L. Sha, and X. Zhu (2006): "Adaptive control of multe–tiered web applications using queuing predictor." In *Proc.* 10^{th} *IEEE/IFIP Network Operation & Mangement Symp.* Vancouver, Canada.

Low, S., F. Paganini, and J. Doyle (2002): "Internet congestion control." *IEEE Control Systems Magazine*, **22:1**, pp. 28–43.

Low, S. H., F. Paganini, J. Wang, and J. C. Doyle (2003): "Linear stability of TCP/RED and a scalable control." *Computer Network*, **43:5**, pp. 633–647.

Lu, C., T. F. Abdelzaher, J. A. Stankovic, and S. H. Son (2001): "A feedback control approach for guaranteeing relative delays in web servers." In *In IEEE Real-Time Technology and Applications Symposium*, pp. 51–62.

Lu, C., Y. Lu, T. Abdelzaher, and J. S. S. Son (2006): "Feedback control architecture and design methodology for service delay guarantees in web servers." *IEEE Transactions on Parallel and Distributed Systems*, **17:9**, pp. 1014–1027.

Lu, Y., T. Abdelzaher, C. Lu, L. Sha, and X. Liu (2003): "Feedback control with queuing–theoretic prediction for relative delay guarantees in web servers." In *Proc. 9th IEEE Real–Time and Embedded Technology and Application Symp. (RTAS'03)*. Toronto, Canada.

Madnick, S. (1969): "Time–sharing systems: Virtual machine concept vs. conventional approach." *Modern Data Systems*, **2:3**, pp. 34–36.

Moruzzi, C. and G. Rose (1991): "Watson Share Scheduler." In *Proc. of the Fifth Large Installation Systems Administration Conf. (LISA '91)*, pp. 129–133. USENIX, San Diego, USA.

Netcraft (2009): "Web server survey." http://news.netcraft.com/.

Noguahi, M. S. S. and J. Oizurnih (1971): "An analysis of the M/G/1 queue under round–rubin scheduling." *Operations Research*, **19:2**, pp. 371–385.

Ohlin, M. (2006): "Feedback Linux scheduling and a simulation tool for wireless control." Licentiate Thesis ISRN LUTFD2/TFRT--3240--SE. Department of Automatic Control, Lund University, Sweden.

Ohlin, M. and M. A. Kjær (2007): "Nice resource reservations in Linux." In *Proceedings, Second IEEE International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'07)*, pp. 20–26. Munich, Germany.

O'Neill, R. W. (1967): "Experience using a time-shared multiprogramming system with dynamic address relocation hardware." In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, pp. 611–621. Atlantic City, New Jersey.

Padala, P., K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant (2009a): "Automated control of multiple virtualized resources." In *EuroSys '09: Proceedings of the 4th ACM European Conference on Computer Systems*, pp. 13–26. ACM, New York, NY.

Padala, P., M. Uysal, A. Merchant, X. Zhu, S. Singhal, and K. Shin (2009b): "Performance differentiation for multi–port arrays: A control–theoretic approach." In *Proc. Fourth International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'09)*. San Francisco, CA.

Paganini, F., J. Doyle, and S. Low (2001): "Scalable laws for stable network congestion control." In *Proc. IEEE 40th Conf. Decision and Control (CDC01)*, vol. 1, pp. 185–190. Orlando, Florida.

RUBiS (2009): "Rice University Bidding System." http://rubis.ow2.org/.

Sayre, D. (1969): "Is automatic folding of programs efficient enough to displace manual?" *Communications of the ACM*, **12:12**, pp. 656–660.

Sharma, S. and D. Tipper (1993): "Approximate models for the study of nonstationary queues and their applications to communication networks." In *Proc. IEEE Int. Conf. Communications*, vol. 1, pp. 352–358. Geneva.

Sharma, V., A. Thomas, T. A. K. Skadron, and Z. Lu (2003): "Power–aware QoS management in web servers." In *Proc. 24th IEEE Real-Time Systems Symp. (RTSS'03)*, pp. 63–72. Cancun, Mexico.

Shils, A. J. (1968): "The load leveler." Technical Report RC 2233. IBM Research.

Slotine, J.-J. and W. Li (1991): *Applied Nonlinear Control*. Prentice-Hall.

Stoica, I. and H. Abdel-Wahab (1995): "Earliest Eligible Virtual Deadline First : A Flexible and Accurate Mechanism for Proportional Share Resource Allocation." Technical Report. Norfolk, VA, USA.

Strachey, C. (1959): "Time sharing in large fast computers." In *Proc. Int. Conf. Information Processing, UNESCO*, pp. 336–341. Paris, France.

Tanenbaum, A. S. (1996): *Computer Networks*, third edition. Prentice Hall, Upper Saddle River, NJ.

Tipper, D. and M. K. Sundareshan (1990): "Numerical methods for modeling computer networks under nonstationary conditions." *Selected Areas in Communications, IEEE Journal on*.

VMware (2009): "Vmware." http://www.vmware.com.

Waldspurger, C. A. and W. E. Weihl (1995a): "Lottery Scheduling: Flexible Proportional-Share Resource Management." In *First Symp. on Operating Systems Design and Implementation (OSDI)*, pp. 1–11. USENIX Association.

Waldspurger, C. A. and W. E. Weihl (1995b): "Stride Scheduling: Deterministic Proportional-Share Resource Mangement." Technical Report MIT/LCS/TM-528. Massachusetts Institute of Technology, MIT Laboratory for Computer Science.

Wang, W., D. Tipper, and S. Banerjee (1996): "A simple approximation for modeling nonstationary queues." In *proc. IEEE INFOCOM '96*, pp. 255–262. San Francisco, CA.

Wang, Z., X. Liu, A. Zhang, C. Stewart, X. Zhu, T. Kelly, and S. Singhal (2007): "AutoParam: Automated control of application-level performance in virtualized server environments." In *Proc. Second IEEE Int. Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID'07)*, pp. 2–7. Munich, Germany.

World Wide Web Consortium (2009a): "Change history of W3C httpd." http://www.w3.org/Daemon/Features.html.

World Wide Web Consortium (2009b): "Status of the CERN httpd." http://www.w3.org/Daemon/.

Xen (2009): "User's manual." http://www.cl.cam.ac.uk/research/srg/netos/xen/readmes/user/user.html.

Xu, W., X. Zhu, S. Singhal, and Z. Wang (2006): "Predictive control for dynamic resource allocation in enterprise data centers." In *Proc.* 10^{th} *IEEE/IFIP Network Operation & Mangement Symp.* Vancouver, Canada.