Diss. ETH No. 13636

A framework for network-aware applications

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Jürg Bolliger
Dipl. Informatik-Ing. ETH
born March 31, 1969
citizen of Gontenschwil, Switzerland

accepted on the recommendation of
Prof. Dr. Thomas Gross, examiner
Prof. Dr. Bernhard Plattner, co-examiner

2000

Seite Leer /
Blank leaf

Abstract

Today's best-effort Internet infrastructure is well-known for its heterogeneity (both in end-system capabilities and connectivity) and the significant fluctuations in service quality that can be observed. These two properties are often problematic from the viewpoint of a networked application, because they often result in unpredictable application behavior. *Network-aware applications* provide an approach to mitigate these problems: they dynamically adapt their demands to match the varying supply of network resources, e.g., with the goal to achieve predictable response times. Such *network-aware* applications must often trade network resources for some *measure of quality* of the data delivered.

This dissertation puts forth the claim that dynamic adaptation is an attractive and often necessary means to achieve the goal on predictable service quality. Unfortunately, network-aware applications are notoriously difficult to construct and evaluate. Two recurring issues need to be addressed when constructing such applications: how to find out about network resource availability and dynamic changes thereof, and how to adapt application behavior (to such dynamic changes), so that a goal on the response time and the quality of the data delivered can be met. Addressing these questions the dissertation makes three main contributions.

First, the dissertation presents a reusable framework for network-aware applications. In contrast to other work, an integral view on the quality-response time tradeoff is adopted, which means that applications built on the framework try to be smart about how to reduce the quality of the data delivered, so that a user-specified time limit can be met, and so that the negative impact on the overall service quality is minimized. Applications are free to define their notion of quality. The dissertation shows that a framework-based approach to the construction of such network-aware applications allows for reuse of the core adaptation process and can therefore shield developers from many of the complexities in dealing with network dynamics. Reusability is achieved by factoring out three aspects of application-specific functionality: the data types handled by the application, the algorithms applicable to achieve quality reductions for a particular type, and the application's notion of quality.

Second, a systematic approach to the evaluation of the complex dynamic behavior of network-aware applications is presented. There are three questions to ask of such adaptive applications. Does adaptation work, and if so, what are the key factors that effect the application's ability to adapt to the network environment? Does network-aware delivery provide a benefit to the user? At what costs can such benefits be obtained? The evaluation establishes that adaptation is robust with respect to many of the parameters that influence adaptation decisions. We find that network-aware content delivery can provide predictable response times over a wide range of bandwidths and CPU powers, can be smart about how to tradeoff quality for network resources, and incurs only small overheads.

Third, the dissertation shows that the performance of network-aware applications depends on the accuracy and timeliness of information about network resource availability. Comparing different alternatives to the task of gathering information about network status, the dissertation demonstrates that the need for accurate and timely information about network resource availability can both effectively and efficiently be satisfied with transport-level monitoring. In addition, our evaluation indicates that the ability to predict bandwidth depends considerably on the transport protocol used. Our prototype monitoring system demonstrates that the implementation of such a monitoring scheme requires only minimal changes to existing protocol stacks and advocates a simple widening of the application programming interface.

Kurzfassung

Viele verteilte Anwendungen, die “best-effort” Netzwerke wie das heutige Internet als Transportmedium benutzen, leiden unter den zuweilen grossen Unterschieden und starken Schwankungen in der Netzwerk Service Qualität, insbesondere der verfügbaren Bandbreite. *Adaptive Applikationen* versuchen hier Abhilfe zu schaffen, indem sie ihr Verhalten dynamisch der Verfügbarkeit von Netzwerk-Ressourcen anpassen, um dem Benutzer eine tolerierbare Service Qualität zur Verfügung zu stellen—zum Beispiel durch Einhalten einer vom Benutzer festlegbaren Antwortzeit. Zu diesem Zweck müssen oft Abstriche in der *Qualität* der ausgelieferten Daten in Kauf genommen werden.

Diese Dissertation zeigt, dass dynamische Anpassung an die Gegebenheiten in Netzwerken ein attraktiver und häufig notwendiger Mechanismus darstellt, um vorgegebene Antwortzeiten einhalten zu können. Leider entpuppen sich sowohl die Konstruktion als auch die Evaluation von adaptiven Anwendungen als ausserordentlich schwierige Aufgaben. Bei der Entwicklung solcher Anwendungen gilt es im Wesentlichen zwei Aspekte zu berücksichtigen: wie erhält die Anwendung Informationen über die Verfügbarkeit von Netzwerk-Ressourcen, und wie kann die Applikation ihr Verhalten anpassen, so dass zum Beispiel eine vorgegebene Antwortzeit eingehalten werden kann. Diese Dissertation leistet die folgenden drei Beiträge.

Erstens stellt die Dissertation ein wiederverwendbares Framework für Anwendungen zur Verfügung, die sich an Veränderungen der verfügbaren Bandbreite anpassen wollen. Im Gegensatz zu verwandten Arbeiten wird versucht, die Qualität der ausgelieferten Daten so anzupassen, dass eine benutzer-definierte Zeitlimite eingehalten werden kann und die negativen Auswirkungen auf die Qualität minimiert werden können. Dabei ist es der Anwendung überlassen, den Begriff Qualität zu definieren. Die Dissertation zeigt, dass ein framework-basierter Entwicklungsansatz für adaptive Anwendungen geeignet ist, da die Kernprozesse der Anpassung wiederverwendet werden können. Die Entwicklung neuer Anwendungen wird somit vereinfacht, da sich die Entwickler nicht um die komplexen dynamischen Eigenschaften von Netzwerken kümmern müssen. Die Wiederverwendbarkeit des Frameworks wird erreicht durch Faktorisierung von anwendungs-spezifischer Funktionalität: die Anwendung definiert die verfügbaren Datentypen, die Algorithmen, welche Qualitätsreduktionen erwirken können, sowie den Begriff der Qualität der auszuliefernden Daten.

Zweitens wird ein systematisches Vorgehen zur Evaluation von adaptiven, verteilten Anwendungen vorgestellt, das es erlaubt, die drei wichtigsten Fragen, die an solche Systeme gestellt werden quantitativ zu beantworten. Können adaptive Anwendungen die vorgegebene Zeitlimiten erfüllen, und falls ja, welches sind die Hauptfaktoren, welche die Anpassungsfähigkeit einer Anwendung beeinflussen? Sind adaptive Anwendungen in der Lage, einen für Benutzer messbaren Nutzen zu generieren? Was ist der Preis, der dafür bezahlt werden muss? Die Eva-

luation zeigt, dass dynamische Anpassung an die Netzwerkumgebung möglich ist und zudem robust ist gegenüber vielen der äusseren Einflüssen, die das Anpassungsverhalten bestimmen. Im weiteren lässt sich feststellen, dass vorgegebene Zeitlimiten recht gut eingehalten werden können, die Qualitätsanpassungen auf eine Art vorgenommen werden, welche die Präferenzen der Anwendung berücksichtigen und dass die Kosten (Rechenaufwand) bescheiden sind.

Drittens veranschaulicht die Dissertation, dass die Anpassungsfähigkeit von solchen Anwendungen von der Genauigkeit und der Aktualität der Informationen abhängt, die sie über die Verfügbarkeit von Netzwerk-Ressourcen erhalten. Der Vergleich von verschiedenen Ansätzen der dynamischen Bandbreitenmessung zeigt, dass die Anforderungen an Genauigkeit und Aktualität der Informationen mit sogenanntem "transport-level monitoring"—einer Kooperation von Transportprotokoll und Anwendung—effizient erfüllt werden können. Zudem zeigt sich, dass die Fähigkeit, die verfügbare Bandbreite abzuschätzen, stark von der Wahl des Transportprotokolls abhängt. Die Realisierung eines Prototypen demonstriert, dass sich ein solcher Ansatz mit lediglich kleinen Änderungen an existierenden Protokollimplementationen umsetzen lässt und dass es daher nützlich wäre, die Schnittstelle zum Betriebssystem um die entsprechende Funktionalität zu erweitern.

Acknowledgments

This work would not have been possible without the help and encouragement of many people.

First and foremost, I thank my advisor, Professor Thomas Gross, for his guidance and counsel. His support and encouragement encompassed many facets that comprise (but are by no means limited to): convincing me to pursue a Ph.D. in the first place, debugging my (first) papers, and finally getting me started with writing up the dissertation. I much appreciated the freedom I was granted and the support I received to pursue my ideas. Despite his extremely busy schedule, Thomas always found time to provide me with valuable and concise feedback. I also thank him for the stimulating working environment he created and the many inspiring exchanges with other researchers he enabled (retreat, conferences, etc.).

I am very grateful to Professor Bernhard Plattner for agreeing (on a fairly short notice) to be my co-examiner. I would like to thank him for closely scrutinizing and questioning my assumptions and for his suggestions that helped me improve the presentation of this dissertation.

Hans Domjan, Roger Karrer, and Michael Näf volunteered to go through the ordeal of proofreading this dissertation. Their comments greatly helped to improve the quality of the text. Any errors and omissions that remain are of course mine.

It is a pleasure to acknowledge the members of Thomas Gross' research group, past and present, and to thank them for their contributions to the lively and inspiring (working) atmosphere. The numerous stimulating and fruitful discussions with Peter Brandt, Hans Domjan, and Roger Karrer especially during the initial stages of my project are being well remembered. Thanks to Hans Domjan who helped me get started with my NetBSD kernel hacking and who never got tired of my countless (more or less) technical questions. I am thankful to Roger Karrer, our poet :-), for his contributions to the Chariot server and for co-advising a number of student projects. Thanks to Urs Hengartner not only for co-authoring most of the papers I contributed to, but for teaching me a thing or two about the intricacies of TCP behavior.

I also want to thank my colleagues of the Institute of Computer Systems for creating such an agreeable and fun environment that made my stay here at ETH an unforgettable experience—there was hardly a coffee break without a good laugh and/or heated discussions about "The Institute". I always enjoyed the "Assistentenabend"—especially the rather legendary one—and I hope to be able to join many more. Many special thanks to all of you for the incredibly hilarious "Mach de Dokter mit em Jürg"—it will always remind me of the great times I spent with an exceptional group of people.

The Chariot project set the stage for this dissertation work[1]. The stimulating, yet often controversial discussions I had with its collaborators—Steven Blott, Alex Dimai, Roger Karrer, Professor Hans-Jörg Schek, and Roger Weber—benefited my work considerably.

[1] My work was in part sponsored by ETH Polyprojekt 41-2641.5.

I was in the lucky position to supervise a number of excellent students during their diploma theses and/or their term projects. Daniel Brennwalder and Urs Hengartner did a fine job implementing and evaluating significant parts of a user-level transport protocol. Adrian Berger, Michael Näf, and Patrick Walther worked on earlier versions of the Chariot image server (thus laying the foundations of the proof-of-concept for the framework). Thomas Ammann, Christoph Burkhalter, and Daniel Estermann worked on other aspects of the Chariot system. I am thankful for the insights gained through the work completed by Roger Karrer, Michael Näf, Cyrill Osterwalder, Felix Rauch, and Roland Vögeli, even though the results of these diploma/term projects may not have directly found their way into this dissertation.

I also want to thank those who provided me with an opportunity to use their host(s) as a platform for the experiments with our home-brewed transport protocol: L. Berc (Compaq SRC, Palo Alto), V. Strumpen (MIT, Boston), M. Franz (University of California Irvine), G. Labahn (University of Waterloo, Canada), W. Weck (Abo Akademi Turku, Finland), J. Templ (Universität Linz, Austria), K. Antreich (Technische Universität München, Germany), G. Jäger Universität Bern, Switzerland), and M. Revilla (Universidad de Valladolid, Spain).

The warmest thanks go to my parents, Margrit and Walter Bolliger, and my brother, Felix, for their continuous and generous support and encouragement during the last three decades.

And last, but not least, I am indebted to Ruth for her love and understanding during the past years. She bore the brunt of my self-absorbed immersion in work, with very little to offer in return from my side. Ruth, thanks for being the wonderful person you are! I hope we will be able to climb many more new heights (or passes ;–) together in the future!

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis 6

1.3 Roadmap for the dissertation 7

2 Background **9**

2.1 Greediness 9

2.2 Reservation 11

2.3 Adaptation 12

2.4 Reuse 22

2.5 Summary 25

3 Framework concepts **27**

3.1 Application domain 27

3.2 Service model 28

3.3 Quality 28

3.4 Implications of the service model 29

3.5 Framework structure: Feedback control loop 34

3.6 Summary 35

4 Feedback loop and adaptation **37**

4.1 Overview 37

4.2 When to adapt? 37

4.3 How to adapt? 39

4.4 Modeling response time 41

4.5 Quality-aware decision making 44

4.6 Framework structure 48

4.7 Start-up behavior 51

4.8 Agility 55

4.9 Discussion 57

4.10 Summary 58

5 Framework instantiation **59**

5.1 Sample application: Chariot 59

5.2 Potential for reuse 66

5.3 Summary 73

6 Evaluation **75**
6.1 Evaluation methodology 76
6.2 Does adaptation work? 83
6.3 When does adaptation work? 90
6.4 Communication-computation tradeoff 101
6.5 Costs of adaptation 108
6.6 Utility of adaptation 112
6.7 Accuracy of resource models 119
6.8 Summary 126

7 Dynamic bandwidth estimation **127**
7.1 Introduction 127
7.2 Related work 130
7.3 Qualitative comparison 136
7.4 Summary 141

8 Transport-level bandwidth modelling **143**
8.1 Throughput models 144
8.2 An Internet experiment 146
8.3 Evaluation of models 147
8.4 Timeouts 154
8.5 Evaluation of protocols 156
8.6 Stability 161
8.7 Summary 164

9 Comparison of monitoring approaches **165**
9.1 Monitor design 165
9.2 Information collection 171
9.3 Evaluation 175
9.4 Summary 189

10 Conclusions **191**
10.1 Contributions 191
10.2 Future work 193
10.3 Concluding remarks 194

Chapter 1

Introduction

1.1 Motivation

The growth of the Internet in terms of users has been proverbial in recent years, as has been the increasing interest in using the Internet as a platform for commerce. These trends have been fueled by the massive growth of the World Wide Web (Web) [14] and the large amounts of information and services one can find on it. For these reasons, there has been and will continue to be substantial competition among (Web) service and content providers to attract as large a set of users as possible[1]. Competition takes place along at least two dimensions: the contents provided, and the quality of the services. Moreover, it will be important that new services can be made available to potential users quickly. While there is an extraordinary diversity in terms of services and information to be found in the Internet, there are two fundamental aspects in this competitive situation among content providers that are worth studying in more detail.

First, besides providing the "right" services and information, a key factor to the success of a content provider in terms of attractiveness is whether the provider is able to serve its users at satisfactory levels of (content) quality and performance. Content delivery with the goal of maximizing user satisfaction is called *quality-aware*.

Second, providers are only competitive if they are able to make such quality-aware services available quickly. This requires short development cycles and that the services be easily deployable.

While data and service quality may have many dimensions and may vary widely from application context to application context, a recurring issue—as far as quality-perception is concerned—is the timeliness of the information delivery. In most systems with the "user in the loop", it is very important that the information requested by the user is provided by the system in a *predictable* and *timely* manner [122] (e.g., to support interactivity). In this context, we envision that ultimately the networked user should not only be able to choose the services and content wanted (as advocated by Fox [57] for instance), but should also be in full control to (i) indicate her preferences about how she values the quality of the content to be provided, and most importantly, (ii) to state how long she is willing to wait for the service. In such a world, the providers must be "smart" about how they satisfy individual user requests. In particular, they

[1] E.g., search engines offer services, news sites offer content. In the remainder, we will use the term content provider to refer to both service and content provision.

must be able to deliver as much data that is relevant to the user's information needs as possible within the time frame allotted.

Despite its recent explosive growth and its diversity of content and services, the Internet, and in particular the Web, have remained a rather primitive place so far. "One size fits all" is the rule for servers providing content and services to users. To understand why this approach becomes increasingly problematic and what can be done to provide a more sophisticated service model for networked applications it is illustrative to briefly review some of today's Internet realities.

Internet realities

While there exist a number of techniques that allow a content provider, e.g., operating a Web server, to deal with the burden of high user demands (e.g., server replication, load balancing [137], etc.), there are two issues hampering the provision of a service model which aims at delivering high-quality data *and* ensuring predictable response times, and these issues cannot be resolved by simply supplying more hardware to improve the server infrastructure.

First, Internet *heterogeneity* is almost as proverbial as its growth. Potential clients vary in several dimensions. First of all, the bandwidth, latency and error characteristics of a client's network access can vary substantially. For instance, paths in today's Internet span six orders of magnitude in bandwidth. The speed of a client's network access may range from the bandwidth of wireless modems ($\approx$ 10kbit/s) up to the speed of fiber optics ($\approx$ 10 Gbit/s). These discrepancies in speed are likely to increase. This heterogeneity is problematic for a content provider, as there is not an obvious best way to deliver the information sought by various users. For example, either the data provided is reduced in quality (and hence in volume) to such a degree that even a client with low bandwidth access can receive the data in a timely fashion, or the data is delivered in high quality, so that at least the clients with high bandwidth access to the provider's server experience satisfactory levels of performance. Both solutions have their drawbacks. Either the users with high-speed network access experience lousy information quality that is in no proportion to the investment made into their network infrastructure, or the users have to wait unduly long for the information to trickle in over their low-bandwidth access line. Timely delivery is just one, albeit an important, aspect of the problems incurred by heterogeneity. Furthermore, clients vary in their processing and displaying capabilities. Thus, content providers must also deal with varying display capabilities of their clients, for instance.

Second, *fluctuations* in network service quality, e.g., in bandwidth, are quite common in today's Internet, both in stationary environments (i.e., for clients with wireline access) and—even more so—for clients in mobile environments (wireless access) [158]. In wireless overlay networks [89], quite significant variations in connectivity and network service quality can be observed. Such fluctuations are often due to congestion in the network. Congestion occurs if the aggregate demand exceeds the supply of bandwidth. Most congestion-aware transport protocols (e.g., TCP [79], DECbit [149], or RLM [113]) react to congestion signals, e.g., packet loss or lower throughput, by reducing the sending rate. Such rate reductions, although in the interest of network stability and hence in the mutual interest of all the network users, result in increased transfer times. Since the demand of other network users can fluctuate significantly and is—at best—extremely difficult to predict, transfer times can exhibit highly unpredictable behavior, which may not be acceptable from the viewpoint of a quality-aware application.

Unless bandwidth is abundant, both problems are likely to prevail. This situation implies that with the current "one size fits all" approach, there will always be users who cannot be served at predictable and satisfactory levels of quality. There are basically two approaches to widen the range of users that can be served at predictable levels of quality by overcoming the problems of network heterogeneity and fluctuating bandwidths: *reservation* or *adaptation*.

Reservation

With a reservation-capable network, applications are able to reserve the required resources (e.g., bandwidth) along the path from the server to the client, such that the desired quality level for the information can be guaranteed and can be delivered within a time frame allotted by the user. There are a number of drawbacks that detract considerably from the attractiveness of a reservation-based approach. First, although considerable research efforts have been dedicated to enhancing the current best-effort Internet infrastructure with reservation capabilities [37, 204, 203], such capabilities are far from being widely deployed yet. Second, resource reservation will most likely not come free of charge, and hence reservation may not be a viable option for certain types of applications. Furthermore, not all reservation requests may be admitted at the service levels desired by the applications (e.g., because of scarce resources, or a cost budget). Therefore, even if reservation mechanisms were ubiquitously deployed in the Internet, *dynamic* quality–response time or quality–cost tradeoffs would often become necessary. Dynamic means that the tradeoffs must be achieved at run-time, and cannot be determined statically, e.g., at configuration-time. Such dynamic tradeoffs can only be achieved by *adaptation*, e.g., by adjusting the quality expectations, such that the goal on response time can be met, or by relaxing the requirements on the response time in to obtain the desired level of quality.

Adaptation

The variable supply of network resources suggests that an application must dynamically adapt to the particular capacity of a network path and to the changes in bandwidth supply to provide predictable levels of service quality for a broad range of networked users.

In this context, adaptation means a trading of resources, either for other resources or for some measure of quality of the data delivered. As providers/applications are faced with fluctuations in network resource availability, they may wish to change the way in which they fulfill a certain request for data by adapting the consumption of the constrained resources. A first option for an application to alter its resource consumption is to trade one resource for another. E.g., compressing data before shipping spends processing power to save bandwidth (as proposed for HTTP/1.1 [50] for instance). Techniques that trade one resource for another are transparent to the user because the data delivered to them is the same, whether the tradeoff is made or not. However, such trade-offs may not be sufficient, either because the adaptation potential of such techniques (e.g., lossless compression) does not match the large discrepancies and swings in bandwidth supply observed, or because there may not be enough of the resource traded for (e.g., processing power on resource-poor mobile hosts) to provide adequate service. Therefore, in addition to trading one resource for another, an application that wishes to change its network resource consumption, e.g., to keep the response time within a user-specified limit, must often trade these resources for some *measure of quality* of the data delivered. If such a tradeoff

for quality is necessary, the application ought to be smart about how to reduce quality and try to minimize the negative impact of the tradeoff on the overall service quality—yet, this is a capability that many current adaptive applications lack.

Network-awareness

Applications that can actively deal with network heterogeneity and dynamic changes in network service quality by adjusting their resource demands are called *network-aware*. Networks are just one of the many resources employed by an application. The model of a network-aware application emphasizes the crucial role of the network connection: in many cases, the network is on the critical path, i.e., performance problems in the network are the cause of the degradation of application performance. In general, however, any component in a system can be a bottleneck, e.g., performance may be restricted by transfers across a local bus or from the disks, or by the amount of computation. If application performance is limited by parts other than the network, such an application should rather try to adjust its behavior in response to these other aspects of the system (disk I/O latency, bus bandwidth, etc.). In the context of this dissertation we focus on the concept of *network-awareness* implying that an application's behavior is primarily controlled by the availability of network resources. Our focus on network-awareness does not imply that other aspects of a system that may effect application performance can be ignored. On the contrary, end-system resources may be important if an application wants to trade communication for computation. For example, if an application wants to adjust to network changes by compressing the data delivered, it is important to make sure that the computation overhead is not worse than the network overload.

Challenges

Network-awareness is an attractive approach to widen the range of networked users that can be served at predictable levels of quality (response time and data/service quality). While there have been various efforts by other researchers to use the concept of network-awareness (adaptivity) to extend the range of conditions over which a networked application performs acceptably, in particular in the realm of (real-time) multimedia applications[2], there remain a number of challenges that have hardly been addressed by previous research.

Quality-awareness. Networked services and applications are becoming increasingly complex. Over time, Web pages, for example, changed from simple text documents (possibly containing a few images) to complex multimedia documents composed of various components. Search engines moved from simple text retrieval over image- to speech- or video-retrieval. Interactive, remote simulation that renders multi-dimensional data on a client's computer [98] is another example of such non-trivial applications. Common to such applications is that they would benefit from an integral, user-centric approach to the quality of the delivered data. For instance, it is important to understand the relative importance of the objects within a complex multimedia document to make suitable bandwidth-quality tradeoffs that minimize the losses in quality of the whole document. Yet, such an integral approach to quality is what many of

[2] A detailed survey of related work will be given in Chapter 2.

the current adaptive systems lack. Most of the systems only make "local" and hence often uninformed adaptation decisions. For example, transcoding proxies [59, 70] make adaptation decisions for each object (e.g., image) to be delivered to the client individually—regardless of the object's context. Furthermore, these systems often employ rather static adaptation policies for the adaptation decisions. Their main objective is to accelerate the content delivery for a certain class of resource-constrained clients. Thereby, they often neglect that the bandwidth-quality tradeoff should try to maximize the quality of the content delivered within the bounds on response time tolerated by the user.

System-awareness. The tradeoffs made by network-aware applications which tackle the problems of heterogeneity and bandwidth variations often include other resources, such as processing power, disk storage, etc. Many adaptive systems currently in use, however, do not take these resources (i.e. their availability) into account when deciding how to best match the network resources available. If the application is to respect user-specified time limits, for instance, it is necessary that the application takes a more global view on the resources employed to make a tradeoff, before it decides on the appropriate adaptation strategy.

Application dynamics. Applications that attempt to match the bandwidth volatility in today's networks often exhibit complex dynamic behavior, which is difficult to study thoroughly. As a consequence, even though numerous network-aware applications have been described in the literature, only very few researchers have tried to go beyond an "exemplary" evaluation of their adaptive systems, e.g., to establish which factors effect adaptation performance the most. There have been even fewer attempts at quantifying the (user-perceived) benefits of adaptation. There are two problems that make attempts at quantifying adaptation performance tricky. First, it is difficult to define metrics that quantify content quality, such that the metrics reflect the "value" of the data delivered. Second, the run-time complexity of an adaptive system may prove difficult to track, because there is a large number of factors that can effect its performance.

Application-network coupling. An additional aspect that has barely received attention in the literature is the interaction between applications and network resource discovery mechanisms. Large efforts have been undertaken by the network research community to characterize network performance and to develop mechanisms that allow for dynamic network resource tracking. Similarly, application developers tried to design their adaptive applications for maximal agility. However, no research studied how precise and timely information about network resource availability must be to be useful to a network-aware application. Similarly, a lot of research is devoted to improve the performance of transport protocols, but, the impact of the transport protocol on application behavior, or more directly on the ability to dynamically track the network service quality, has hardly been studied.

Ease of deployment. As the pace at which new applications are made public is ever increasing, there is a clear need for fast *development* and *deployment* of new services. As far as the first issue is concerned, the solutions to the problem of network heterogeneity and network variability adopted by many of the applications currently used are often tailored to the specific needs of the individual application or a specific programming model [205], and there exists no general

approach to develop network-aware applications for other application domains. Yet, a useful set of abstractions, a set of reusable building blocks could be highly effective in shielding developers from the many complexities inherent in network-aware systems and could help shorten the development cycle for new services and applications. When it comes to the issue of deploying a new application, it is important that the infrastructural changes needed to make a new service available to a large user community are kept to a minimum.

Goals

To this end—and tackling these challenges—this dissertation sets out to develop a general approach to the construction of network-aware applications. The goal is to identify a set of abstractions common to a significant class of network-aware applications. These abstractions must allow for a simple, but flexible way to characterize an application's (or user's) notion of quality. Abstractions and mechanisms are sought that facilitate *software reuse* and help a large number of applications to implement a service model that allows to trade off the quality of a service and the time required to fulfill the service. Moreover, applications building on these software structures should be easily deployable by Internet content providers.

In addition, this dissertation work aims at broadening the understanding and at improving the characterization of the complex dynamic behavior of network-aware applications by means of a systematic performance evaluation that explains the key factors effecting the performance of network-aware systems. In this context, emphasis is placed on understanding the importance of on-line bandwidth estimation for the needs of network-aware applications.

1.2 Thesis

The heterogeneity and the volatility in service quality present in the Internet pose a problem to many quality-aware networked applications. I claim that:

- Network-aware content delivery, with the goal to achieve predictable response times at the highest levels of quality possible, is capable of extending the range of network conditions for which acceptable application behavior can be provided. Sender-based adaptation enables the support of an integral approach to service quality and lends itself well to fast deployment.

- A framework-based approach to the development of such network-aware applications is beneficial as it allows for reuse of the core adaptation (i.e., decision making) process and can therefore shield developers from many of the complexities in dealing with network dynamics.

- The performance of network-aware applications, defined as the application's ability to meet a user-specified time limit, depends on the ability of getting accurate and timely feedback about network resource availability.

- Information about network resource availability can be provided both accurately and efficiently by light-weight monitoring at the transport-level. The transport protocol has a notable impact on the accuracy of models used for bandwidth estimation.

1.3 Roadmap for the dissertation

The dissertation establishes the thesis in the following way:

- It argues that Internet heterogeneity and variations in bandwidth continue to prevail, that adaptation is required to meet the goal of predictable content delivery (even if reservation schemes were widely deployed in the future), and that sender-based adaptation best fulfills the goal of easy deployment.

- The dissertation presents the design and implementation of a framework for network-aware applications and addresses adaptation inherent problems such as agility, start-up problems, etc. It draws a line between the framework and application specifics and shows the potential for framework reuse by presenting a sample framework instantiation: Chariot[3], a networked image search and retrieval system.

- Furthermore, the dissertation presents a systematic, application-oriented evaluation methodology based on trace modulation [133] which is applied to the example application to quantitatively validate the claims that

  - Sender-based adaptation is capable of providing acceptable application behavior for a broad range of network conditions, that is, across a broad range of bandwidths and bandwidth volatilities.
  - The performance of network-aware applications, defined as the application's ability to meet a user-specified time limit, depends on the ability of getting accurate and timely feedback about network resource availability.

- The dissertation discusses different approaches to obtain information about network status. It shows that the need for accurate and timely information about network resource availability can easily be met by passing transport protocol information up to the application, and presents results from a large-scale Internet experiment which show protocol effects on the ability to estimate and predict available bandwidth.

The dissertation is organized in two parts: while the first part (Chapters 2–6) concentrates on application-level aspects of network-aware applications, and describes and evaluates the adaptation framework, the second part (Chapters 7–9) deals with issues of application-network cooperation and methods of gathering information about and estimating network resource availability. Chapter 2 discusses related research in reservation-based networks and in network-aware applications and thereby provides evidence that the challenges outlined above have largely remained untackled so far. The chapter also reviews several concepts of software construction that help increase reuse and motivates why frameworks are an interesting option for the type of applications considered here. Chapter 3 introduces the basic concepts of the framework proposed and describes the details of the service model. Chapter 4 then presents the implementation of the framework, details the adaptation process, and discusses adaptation inherent problems, such as agility and the start-up problem. Chapter 5 explains how the framework is instantiated by presenting a sample application, Chariot, that is derived from the framework, and qualitatively

[3] Swiss (**CH**) Analysis and **R**etrieval of **I**mage **O**bjec**T**s

discusses the potential of code reuse by other applications. Chapter 6 introduces the evaluation methodology and presents quantitative results that allow us to identify the key factors effecting adaptation performance and to assess the importance of timely and accurate information on network resource availability. Chapter 7 discusses several approaches to on-line bandwidth estimation. Chapter 8 presents the results of a large-scale Internet experiment used to analyze the efficacy of different transport-level throughput models and briefly discusses issues related to bandwidth prediction. Chapter 9 compares application-level and transport-level monitoring in terms of efficiency as well as timeliness of their bandwidth estimates. Chapter 10 summarizes our findings and concludes the dissertation.

Chapter 2

Background

The goal of this dissertation is to provide a user-centric service model for networked applications that allows to obtain predictable response times while maximizing the "value" of the data delivered to the user. This chapter discusses approaches that can be taken to achieve this goal and concludes that adaptation is an attractive—and often necessary—means to achieve these two objectives. The discussion of related research into network-aware applications reveals the issues that have largely remained open so far and will consequently be explored by this dissertation.

2.1 Greediness

The first class of applications that attempts to achieve predictable response times while maximizing the value of the data delivered builds on the principles of *application level framing*. The concept of application level framing was coined by Clark et al. [38] and states that application semantics should explicitly be included in the design of a (new) protocol to allow for efficient data transfer. The idea is that the application should break the data into suitable aggregates (that can be processed at the receiver independently and possibly out-of-order) and that the lower layers (e.g., the transport protocol) should preserve these frame boundaries as they process the data. These aggregates are called *application data units (ADUs)*. Such a framing allows to completely decouple data manipulation steps (encryption, presentation conversion, etc.), which are the responsibility of the application, and transfer control functions (flow/congestion control, multiplexing, etc.), which are to be provided by the transport protocol. This separation of concerns adds flexibility, e.g., to choose the appropriate transport protocol. For instance, application developers are free to choose a simple, unreliable transport protocol with no transfer control functions (e.g., UDP [176]), as long as it supports datagram semantics.

How can application level framing help in establishing the objectives of predictable response times? By means of a loss-resilient encoding of the data to be transferred. Data can be video, image, text, or other digital media. Each data item to be transferred is encoded and packed into independent ADUs in such a way that (i) each ADU contains information about its place within the sequence of ADUs produced, (ii) each ADU contains data that can be decoded independently at the receiver and (iii) the loss of an ADU does not prevent the (potentially quality-reduced) content of the data item to be reconstructed at the receiver. Given such a loss-

resilient encoding and segmentation of the application data, the application must only choose an appropriate fixed rate at which it paces the packets into the network to achieve a certain time limit. The bandwidth-quality tradeoff will implicitly be made by the network. If bandwidth is scarce, ADUs will be dropped. As a consequence, quality suffers upon reconstruction of the content from the ADUs that have reached the receiver.

Turner et al. [184] were among the first to apply these ideas to the transfer of image data. They proposed a simple, bitmap-based algorithm for encoding images into ADUs, such that the receiver can recover from dropped packets without requiring the sender to retransmit them. Amer et al. [5] present a similar scheme for the encoding of GIF images. Similar ideas have been applied to the encoding of video data. E.g., Albanese et al. [2] describe PET (Priority Encoded Transmission) of MPEG-video. By adding redundancy in the encoding, PET can assure the delivery of a user-specified fraction of I-, P- and B-frames [123], given that the loss-rate does not exceed a certain threshold (used to compute the amount of redundancy needed). More popular, but unpublished encoding algorithms have been incorporated in the commercially available tool RealVideo [151, 153].

Such an approach at achieving predictable response times may seem very attractive from an application developer's perspective because it is simple to implement (given an appropriate data encoding algorithm exists) and because it does not require that the application deals with the dynamics of network resource availability. In fact, such encoding-based transmission at a fixed data rate is becoming increasingly popular as the success of RealVideo and similar continuous streaming services indicates. We call such data transmission schemes *greedy* because the application consumes as much bandwidth as possible and needed for its own purposes.

However, although attractive for a single (selfish) application, such a greedy transmission scheme is highly problematic for the network as a whole. These applications "pump" non-congestion-controlled traffic into the network. That is, their traffic flows (i) are *unresponsive* to incipient congestion, as they do not react to packet loss, and (ii) may therefore consume a *disproportionate* amount of the bandwidth available. Floyd et al. [53] show that unresponsive and disproportionate bandwidth flows can not only be drastically *unfair* to competing TCP (and other congestion-controlled) connections but can eventually even result in an Internet *congestion collapse* (from undelivered packets)[1]. Therefore, such traffic poses a serious threat to the stability of the Internet, which has relied heavily on the end-systems/applications to cooperate and participate in end-to-end congestion control so far [79]. Although routers may in the future move from the simple first come, first served scheduling policy to more sophisticated policies that isolate and "punish" unresponsive flows [54, 165, 53], it is important that new transport protocols or applications employ end-to-end congestion control mechanisms and abide by a few rules that ensure their traffic streams are TCP-friendly [103]. Empirical evidence indicates that TCP friendliness is a sufficient condition to guarantee network stability.

[1] A second type of congestion-collapse, so-called fragmentation-based congestion collapse [53], may occur with the class of applications sketched. If the ADU size does not match (i.e. is larger than) the network path's maximum transmission unit (MTU), ADUs must be fragmented and reassembled at the network layer. If one fragment is lost, the whole ADU is worthless and the remaining fragments of the ADU waste precious network resources [91].

2.2 Reservation

A second approach to achieving predictable response times is to reserve the resources (e.g., bandwidth) needed from a sophisticated network infrastructure that supports multiple traffic classes and resource reservations. The Internet research community has devoted much effort to designing an integrated services Internet architecture[2], which is an architecture capable of supporting real-time applications as well as (best-effort) data applications (see, for example [37, 11, 49, 83, 138, 204] and references therein for a small sampling of the literature in this area). The Internet Engineering Task Force (IETF) recently promoted to "Proposed Standard" level extensions to the Internet architecture that will enable the Internet to support reservations. These reservations manage resources (e.g., bandwidth) that are set aside for a particular traffic stream (see [203, 166, 167, 200, 201] for the relevant RFCs and for additional supporting material). In this architecture, networked applications can still send best-effort packets, but in addition they have the option of requesting a reservation for their traffic streams. To obtain a reservation, an application requests a certain amount (characterized by a traffic specification) and quality (specified by a service specification) of service; the network then decides whether or not it can satisfy this request. While there are many mechanistic differences between the various integrated services proposals, they all share the two fundamental aspects that (i) applications have the ability to reserve bandwidth, and (ii) the network exercises control—known as admission control—over these reservation requests, so that it can ensure the level of service given to reserved traffic.

Provided the ability to reserve bandwidth, a quality-aware application (e.g., faced with a limit on the response time) must merely determine its resource needs, that is, estimate how much data it wants to transfer across the network to satisfy the user's information needs, and then request that the appropriate amount of bandwidth be set aside for its traffic streams. Such an approach, i.e. the delegation of all the complexity needed to ensure a certain service quality to the network, may seem to be an attractive solution from an application developer's perspective, because it promises to keep applications simple and lean. However, reservation is no panacea—for a number of reasons.

First, despite the considerable research efforts, reservation-capable networks are not widely deployed yet. And considering the slow rate at which wide-scale Internet infrastructure changes are taking place, it is likely that in the future some network architectures (or their implementations) may not support reservations at all or may support them only to a limited degree, or that not all sites are willing to invest in the infrastructural upgrade. Second, as network providers attempt to develop usage-based charging schemes [64], there will be financial incentives to restrain applications from uncontrolled use of network resources. Third, not all reservation requests may be admitted at the service levels desired by the applications (e.g., because of scarce resources). Moreover, to support quality-aware applications, end-to-end QoS guarantees may be necessary. This implies that not only network aspects must be considered, but that end-system and operating system resources must also be taken into account [129]. This requirement holds especially for continuous media applications as they have the most stringent resource requirements [172, 161]. As a consequence, resource orchestration may become crucial to allow

[2]In this context, the term service refers to network capabilities offered to applications. The initial proposal by Clark et al. [37] differentiates between *guaranteed*, *predicted* (real-time) and *best-effort* (datagram) service.

for meeting the user's QoS requirements on an end-to-end basis [128, 26].

As a consequence, even though an integrated services network may allow to make bandwidth reservations, and thus may have the potential to simplify application development, an application must still address (at least) the two issues of (i) how to find out what and how much to reserve, and (ii) how to adjust to meet the confirmed reservation, which may be less than the application has asked for. Note that from a software engineering point of view, reservation-based applications and purely adaptive applications that use a best-effort-only network require the same software technology: the application must be able to adjust its resource demands, either to meet a limit imposed by a reservation or to meet some constraints imposed by the network. In either case the application must be adaptive.

Since quality-aware applications may have to be adaptive in any case and because, in fact, a number of current networked applications *are* adaptive (e.g., vat [81], vic [112], ivs [20], etc.), one may question whether the added complexity of enhancing the Internet infrastructure with reservation capabilities is justified by the performance benefits that can be derived from such an infrastructure.

Breslau et al. [24] are the first to take a systematic approach at the debate "reservation versus best-effort (and adaptation)". Using an analytical model, Breslau et al. address the fundamental question whether the Internet should retain its best-effort-only architecture, or whether it should adopt one that is reservation-capable. The differences between the network models are characterized in terms of application performance and total welfare for all applications using the network. In addition to raw network-centric performance numbers, the authors incorporate two elements in their analysis: the utility functions of the application studied (how application performance depends on network service) and the adaptive nature of the applications (how applications react to changing network service). The question cannot be answered conclusively, since it would be necessary to know the future cost of bandwidth and the nature of future traffic load. For some types of traffic load, the differences between the network models depend on the cost of bandwidth and on how much cost the increased complexity of reservations add to the network. However, one of the conclusions of the study is that for certain types of traffic load, adaptive applications (as opposed to rigid, non-adaptive applications) make the case for reservation-capable networks almost vanish.

2.3 Adaptation

The discussion in the previous sections revealed that neither reservation nor loss-resilient data encoding with constant-rate transmission of the encoded data provide the ultimate solution for the problem of supporting a service model that allows to achieve predictable response times. Reservation-capable networks can still call for dynamic adaptation; and non-congestion-controlled transmission of content is not a ("socially") viable option. Therefore, there is clearly a need for network-aware content delivery to cope with the problems of heterogeneity and bandwidth volatility in today's networks. Clearly, adaptation is not a cure to all these problems either, but, we find that it is often an attractive and necessary means to achieve the objectives. In fact, adaptation is often complementary to those other approaches.

There are many different forms of network-awareness. E.g., continuous media applications often perform delay (or play-out point) adaptation to ensure smooth reproduction of the

Media type	Specific encodings	Distillation axes
Text	Plain, HTML, RTF, Postscript, PDF	Richness (formatting), Content (summary, headings)
Image	GIF, JPEG, TIFF, PPM, XBM	Resolution, color depth, color palette, requantization
Audio	MPEG, MIDI, WAV	Low-pass filtering
Video, Image Sequence	MPEG, H.261, NV, M-JPEG	Frame rate, selective frame dropping, hierarchical filtering,

Table 2.1: Media types and distillation dimensions.

audio/video signal received (e.g., vat [81], RealAudio [151]). Other applications hide client heterogeneity by trading performance for the levels of security with which the data is encoded [160]. Among the countless different forms of network-aware content delivery we concentrate on techniques that trade *bandwidth* for some *measure of quality* of the data delivered and/or for processing power (to respect a user-specified time limit). As a result of the discussion in Section 2.1, we concentrate on content delivery that is controlled by a congestion-aware transport protocol. In this context, adaptation implies the ability to produce a range of variations (or versions) for any object to be transferred across the network so that an appropriate version can be chosen to match the current network environment. Such quality adaptation is often called *transcoding* or *distillation*. The two terms designate the transformation that converts a (multimedia) object from one form to another, trading off object fidelity for size. Clearly, such transformations are media type specific. Table 2.1 gives a partial list of possible adaptation strategies for commonly used media types. Mohan et al. [124] note that "modality changes" provide further attractive distillation axes for complex multimedia objects. E.g., a video stream could be transcoded into a few key frames representing different scenes from the video; or voice could be converted to text to achieve space savings.

There is a fairly large and rapidly growing literature of approaches based on concepts of network-aware adaptation. This section classifies the approaches to network-aware content delivery taken by other researchers. It then turns to discuss whether and how (well) these approaches address the challenges listed in Chapter 1. Thereby we defer discussion of reusability aspects to Section 2.4.

The different approaches to network-aware content delivery can be classified according to various criteria. First, which party in a networked application is responsible for (i.e., initiates) the adaptation: the sender, the receiver(s), or intermediary nodes in the network (e.g., proxies)? Second, which "layer" in the adaptive application is responsible for resource discovery and for making the adaptation decisions: the operating system, the application, a collaboration of system and application, or the user? Third, what is the main objective of the adaptation: to tide networked applications over bandwidth fluctuations, network heterogeneity, or heterogeneity in client capabilities? There are further criteria, such as the communication paradigm used (unicast vs. multicast) or the media types supported. Table 2.2 gives an overview of the related work discussed in this section, classified according to the three criteria above. The last column lists specific attributes on communication paradigms or media types, if they deviate from the "default", which comprises unicast communication and support for multiple media types.

System	Citation	Where to adapt?	Who decides?	Objective[a]	Remarks
Odyssey	[130]	Receiver	Sys./App.	BW-F	
Prayer	[16]	Receiver	Sys./App.	BW-F	
RLM	[113]	Receiver	Application	BW-F/H	Multicast, Video
TOMTEN	[39]	Receiver	User	BW-F/H	
GloMop	[59]	Proxy	Application	CL/BW-H	
(Han et al.)	[70]	Proxy	Application	CL-H	Image
(Chandra et al.)	[34]	Proxy	Application	CL-H	Image (JPEG)
(Hemy et al.)	[72]	Proxy	Application	BW-F/H	Audio/Video
HIPPARCH	[93]	Sender	System	BW-F	
InfoPyramid	[124]	Sender	Application	CL/BW-H	
RAP	[152]	Sender	Application	BW-F/H	Video

[a]Legend: BW = bandwidth, CL = client; H = heterogeneity, and F = fluctuations

Table 2.2: Overview of the related work reviewed.

2.3.1 Receiver-initiated adaptation

Odyssey. To cope with the turbulences of mobile environments, mobile clients accessing remote data must dynamically adapt their behavior by trading the quality of fetched data for the speed of fetching it. Noble et al. [132, 130] claim that such adaptation is best provided by *application-aware adaptation*—a collaboration between the operating system and its applications—at the receiver. In this collaboration, the operating system is responsible for providing the mechanisms for triggering when to adapt, while applications are free to set adaptive policies that define how to adapt. The Odyssey prototype [132] provides a central authority responsible for (network and client) resource tracking, registration of application resource needs with the system and an upcall-mechanism [35] which notifies the application when significant changes in resource supply have been detected by the monitoring subsystem. The collaborative effort between system and applications is well suited to support application concurrency on resource-constrained mobile devices as it allows for fair (or at least controlled) partitioning of resources among competing applications. Noble et al. modified three existing applications to make use of the Odyssey system [132]: a public-domain video player (xanim), a Web browser (netscape), and a speech recognizer (Janus). In the Odyssey system, adaptation decisions are made at the client and are based on ranges of availability. The range (or window of tolerance) expresses the application's resource expectations for "normal" operation, as well as the application's desire to be notified if the availability of the resource strays outside the window.

Prayer. In Prayer [16], Bharghavan et al. use ideas similar to those present in Odyssey. They also postulate the collaboration of system and application by splitting adaptation policy (application) and adaptation mechanisms (system). In Prayer, resource availability is monitored by a central authority and adaptation decisions are based on (multiple) ranges of availability. The application can specify multiple "QoS classes" (the analogon of Odyssey's window of tolerance) and associate with each QoS class a handler that implements the application's policy for the specified resource range. The system monitors resource availability, determines the appropriate QoS class and calls the respective handler if a change occurs. The authors stipulate that

decisions be made at the client/receiver and assume that there is a consistency mechanism that notifies the server (or proxy) of any QoS class change.

RLM. McCanne et al. [113, 109] pioneered scalable transmission of multicast video data. The authors devised a layered video encoding [115] that is used to selectively forward the different encoding layers on multiple IP-multicast groups. For scalability reasons, the adaptation is receiver-driven. Receivers are responsible to deal with heterogeneity and fluctuations in the transmission rate. By joining a subset of the multicast groups they specify their subscription level. This scheme only allows for coarse-grain congestion control, because it is limited by the (somewhat slow) speed at which changes of multicast group membership can be propagated. A receiver drops layers of the video signal if it observes packet loss. From time to time, it probes availability of additional bandwidth by adding a new layer. The application is in full control of the adaptation decisions.

TOMTEN. De Silva et al. [39] propose a completely user-centric adaptation scheme. The user makes adaptation choices at application start and whenever she becomes dissatisfied with the quality of the service received. Upon user intervention the TOMTEN system *reacts* by discovering the available resources, assessing the change in available resources since its last invocation, and by determining possible adaptation strategies that are then presented to the user. With TOMTEN there is no need to change applications to benefit from the reactive framework. Application calls to/from the network are intercepted and redirected either through adaptation modules that may filter the application data stream or through a different network interface. Filtering is often performed at a proxy. Their filter implementation supports the JPEG and MPEG formats. The TOMTEN system performs no continuous monitoring, since the adaptation mechanism is completely reactive.

2.3.2 Proxy-based adaptation

GloMop. GloMop stands for global mobile computing by proxy. Fox et al. [58, 59] pioneered on-the-fly adaptation by transformational proxies. These proxies host datatype-specific distillation services that can be invoked to dynamically adapt the quality of the data delivered. The authors claim that pushing adaptation into the network infrastructure is a widely applicable, cost-effective, and flexible solution to addressing the problems of client heterogeneity. The solution is widely applicable because it supports various applications and because there is no need to change server- or client-side applications. A proxy appears as client to servers and as server to clients. The solution is cost-effective because it allows for incremental deployment of proxies when scalability becomes a concern (see Section 2.4 for details). Some of the ideas described by Fox et al. [58, 59, 61] have been commercialized (mostly to adapt Web content for display on personal digital assistants (PDAs), such as the 3Com PalmPilot): Proxynet [147], Intel's Quick Web [148], AvantGo [8], and possibly others.

Despite its success—as indicated by the large number of followers—the proxy-based transcoding proposed by Fox et al. [58, 59] exhibits a number of shortcomings. The main objectives are adaptation to client heterogeneity and acceleration of Web content delivery. Variability in bandwidth availability is not taken into account. The transcoding proxies often consider a few

types of client devices and employ static, ad-hoc content adaptation strategies. In particular, they do not consider the time required to transcode the media objects (which among other factors depends on the proxy workload). Furthermore, they simply assume that the connection between server and proxy is high-bandwidth, which turns out to be an over-simplification in reality [131]. As a result, proxy-based transcoding cannot make a sophisticated quality-time tradeoff and may even fail to reduce response time. Recent work on transcoding proxies tries to remedy some of these shortcomings (as described in the following two paragraphs).

(Han et al.). Han et al. [70] present an analytical framework which describes when to transcode and when not to transcode for proxies. The idea is that transcoding is only performed when response time is reduced. Their analysis is based on models for the transcoding time, the size of transcoded images, and on accurate predictions of both server-proxy and proxy-client bandwidth. This analysis is used to derive decision heuristics that decide when and how to adapt. The heuristics are adopted in the actual implementation. It is important to note that these heuristics are static adaptation policies; the proxy does not use the models to dynamically derive adaptation decisions. The authors also describe (but did not implement) automated transcoding, a process by which the proxy adapts its image coding to network variability while trying to meet an upper bound on the delay tolerated by the end user. The actual implementation of an HTTP-proxy adapts to client heterogeneity, image content, and user preferences.

Addressing the lack of system-awareness in previous work, Han [69] studies the impact of a mobile client's processing speed on the adaptation decisions at a proxy. Based on the observation that PDAs can have very low processing speeds, Han compares three alternatives of proxy-based image transcoding: (i) the browser at the client must scale and decompress image content; (ii) the proxy pre-scales, and the browser decompresses; (iii) the proxy pre-scales, decompresses and transmits a grayscale bitmap, the browser merely displays the image. The study concludes that proxies which take the CPU limitations of their clients (PDAs) into account can dramatically reduce the end-to-end latency for an image download by migrating some or all CPU-intensive tasks from the slow PDA to the transcoding proxy.

(Chandra et al.). Examining previous work's ad-hoc adaptation decisions that barely considered quality aspects in their tradeoffs for bandwidth, Chandra et al. [34] postulate that "informed transcoding techniques" help to balance the need for good quality of multimedia content while reducing consumed network bandwidth. By "informed" the authors mean that a transcoding proxy must take media specifics into account and must consider "image information quality" (as analyzed by Ford [55]) when making adaptation decisions. As far as media specifics are concerned, they carefully analyze the JPEG image format and find that the "input quality" of an image is an important factor that effects transcoding costs and output size (these findings are consistent with the results reported by Walther in a study done at ETH Zürich [189]). If the input quality is not taken into account, uninformed transcoding decisions may lead to the transcoded image being larger than the original image.

(Hemy et al.) The proxy-based approaches mentioned so far only adapt content delivery to client heterogeneity. Hemy et al. [72] devised a transcoder that adapts the bandwidth of MPEG system streams containing both video and audio signals to also account for fluctuations in net-

work bandwidth. They employ smart selective frame dropping to ensure that quality distortions of the MPEG stream displayed at the receiver are kept to a minimum. Note, their transcoder may also be co-located with the server (but it still acts as a proxy from the client's point of view).

2.3.3 Sender-initiated adaptation

HIPPARCH. Knutsson et al. [93] employ lossless data compression at the transport layer, i.e., in the TCP protocol stack, to maximize "user-perceived throughput", i.e., to minimize the transfer time. By virtue of using lossless compression the adaptation is transparent to the application. The adaptation scheme assumes that all CPU resources are available to perform compression. Their system dynamically adapts the compression ratio to match the transmission rate and processing speed. The adaptation decisions are based on buffer occupancy at the sender. Empty buffers indicate that the sender cannot keep up with the network, therefore, the compression ratio is reduced. On the other hand, if buffers build up, the network is too slow and the sender reacts by increasing the compression ratio. If the receiver cannot keep up, the buffers at sender fill up (due to TCP's flow control).

InfoPyramid. The system described by Mohan et al. [124] tries to adapt multimedia Web content to optimally match the capabilities of client devices. They employ two (application-level) concepts. First, the *InfoPyramid* is a multi-modal, multi-resolution representation hierarchy for digital media data, which—besides the commonly used distillation mechanisms (see Table 2.1)—allows for progressive content delivery, video-to-image sequence, image-to-text, or speech-to-text transcoding. Second, there is the *customizer*, which selects a content representation that meets the capabilities of a particular client and that maximizes the "value" delivered to the user. The authors introduce an application-independent concept to characterize the "value" of content, which is based on the rate-distortion theory by Shannon [164]. While being an interesting approach because of its application independence and because it allows for a formalization of the optimization problem, the authors note that the characterization may not always correspond with user perception.

RAP. As an example for the multitude of video applications that use sender-based adaptation, we review the work of Rejaie et al. [153, 152]. In [153], the authors describe an end-to-end TCP-friendly [103] rate adaptation protocol (RAP), which is suited for unicast playback of realtime streams and for other semi-reliable rate-based applications. In contrast to RLM, it exercises fine-grained unicast congestion control. Rejaie et al. [152] present a mechanism for using layered video transmission (e.g., [115]) in the context of unicast congestion control. Their quality adaptation mechanism adds and drops layers of the video stream to perform long-term coarse-grain adaptation, while using a TCP-friendly congestion control mechanism to react to congestion on very short timescales. The mismatches between the two timescales are absorbed by using buffering at the receiver.

2.3.4 Discussion

This section discusses the related work presented in the previous sections according to the two main criteria used for their classification: who initiates the adaptation process (where to adapt): receiver, proxy, or sender? who decides: system, application, collaboration of system and application, or the user? The different approaches are reviewed according to the five issues introduced in Chapter 1. We claim that the issues of quality-awareness, system-awareness, application dynamics, application-network integration (resource tracking), and ease of deployment have largely remained unchallenged by related research so far. Thereby, we first address fundamental benefits and problems (e.g., of a proxy-based architecture) before we point out strengths and weaknesses of specific approaches.

Where to adapt?

Proxy. One big benefit of proxy-based adaptation is that it is totally transparent to the content providers; they do not have to change the way they author or serve content. In addition, the proxy approach is also transparent to the client; the client application need not be changed. Because of its transparency, the proxy approach allows for incremental deployment, which is a cost-effective solution to scalability problems [57]. The flexibility offered by a proxy-based approach has also been recognized in a variety of other application contexts, especially in the mobile environment [202].

On the other hand, there are a number of drawbacks to this approach: First and foremost, a transcoding proxy can commonly make only very limited quality-aware decisions. E.g., it can transcode an image such that the image delivered fits with the client's displaying and processing capabilities and such that the delivery of that particular image is accelerated. However, the proxy often has no knowledge about the context of the particular image (is it important or even visible within the complex Web page requested or not?), and thus must make a rather uninformed "local" adaptation decision. In fact, most proxy-based approaches currently in use deal only with client heterogeneity; they concentrate on a few static adaptation policies. As a consequence, proxies are hardly suited to make response time-quality tradeoffs for complex applications. Second, with a proxy approach, content providers have no control over how their content will appear to different clients. Third, for a proxy to make data-specific distillation decisions it must be able to analyze the content or structure of the data through the proxy (see [124] for a detailed critique on this aspect). Furthermore, a proxy-based approach must track the bandwidth of both the server-to-proxy and proxy-to-client connection to make sound adaptation decisions as argued by [131]. Other aspects, such as the question where to place proxies, or how to (dynamically) compose the services provided by proxies, are subject to ongoing research [6].

Receiver. Burdening the receivers with the task to find good bandwidth-quality tradeoffs has the big benefit of a solution that is scalable. Servers do not have to process feedback from and make adaptation decisions for the many clients that request network-aware content delivery. A second aspect that is in favor of receiver-initiated adaptation is that—compared to proxy-based adaptation—it is better suited to make informed (or "global") adaptation decisions that are based on the context of the object to be transcoded [130]. Receiver-initiated adaptation can

therefore better opt for an integral response time-quality tradeoff than a proxy-based approach. (The same applies to sender-initiated adaptation when compared to a proxy-based solution). Furthermore, a receiver-based approach only needs to monitor one connection, namely the one to the server.

The flip side of receiver-initiated adaptation is that the receiver merely initiates the adaptation, but the actual transcoding necessary to achieve a bandwidth-quality tradeoff must be carried out somewhere else, e.g., at the server or at a proxy. There are two options to deal with this situation, both of which are problematic: First, if applications are to choose among different (quality-reduced) versions of the original media object stored at the server/proxy, the client must know which versions exist at the peer. Such a solution may require an inordinate amount of coordination with the peer. Second, the client may request that the versions are dynamically distilled at the server/proxy. To make an informed decision about the quality of the object to be delivered the client would have to take the distillation costs into account. The costs depend on the resource availability at the peer, which is often beyond the control or even the knowledge of the client.

Sender. The pros of sender-initiated adaptation are that the content provider is in full control of the content quality and can make informed decisions that take the context of objects to be transcoded into account. Therefore, the sender can more easily achieve a response time-quality tradeoff than the other alternatives. Elaborate coordination with the client is not necessary. Moreover, CPU resource availability at the client is likely to fluctuate less strongly as at the server[3]. Thus, the sender-based approach can more reliably make tradeoffs that incorporate client capabilities. Again, only one connection needs to be monitored.

On the down side, we note that sender-based approaches do not scale as well as the receiver-based solutions. In addition, server applications have to change the way how they serve content.

Table 2.3 summarizes the discussion of the three alternatives.

Who decides?

System. System-only decision making is totally transparent to the application (Noble [130] names this type of adaptation *application-transparent*). On the positive side, this means that no application modifications are required. Furthermore, since the OS takes care of resource tracking, there is a central point of control for all concurrent resource-aware applications. Hence, the system can enforce resource allocations and employ resource optimizations across multiple concurrent applications. System-controlled monitoring and adaptation is best suited to ensure high *agility*. An agile system is a system that reacts swiftly to changes in resource availability.

On the down side, application-transparency also means that the system has no knowledge of application specifics and is therefore not in a position to make application-specific, quality-aware adaptation decisions. In fact, it cannot perform a bandwidth-quality tradeoff. Therefore, such an approach is hardly suited to support application diversity. Furthermore, the lack of knowledge about application resource needs may result in an under-utilization of the adaptation

[3]Dinda [41] reports that the load on desktop machines, which typically act as clients, varies less strongly in absolute terms than the load on other machines, such as compute servers or production clusters.

<table>
<tr><th></th><th>Proxy</th><th>Receiver</th><th>Sender</th></tr>
<tr><td rowspan="4">Quality-awareness</td><td>– "local" decisions</td><td colspan="2">+ "global", quality-aware decisions</td></tr>
<tr><td>– no time limit</td><td colspan="2">+ integral time-quality tradeoff</td></tr>
<tr><td>– must analyze content</td><td>– coordination with peer</td><td rowspan="2">+ full control over content quality</td></tr>
<tr><td colspan="2">– no quality control by content provider/author</td></tr>
<tr><td>System-awareness</td><td>+ can incorporate client CPU availability</td><td>– no control over peer CPU resources</td><td>+ can incorporate client CPU availability</td></tr>
<tr><td>Dynamics</td><td colspan="3">not applicable</td></tr>
<tr><td>App./Net. coupling</td><td>– must track bandwidth of two connections</td><td colspan="2">+ only one connection to monitor</td></tr>
<tr><td rowspan="2">Ease of deployment/ development</td><td>+ transparent to content provider and client</td><td>– must change client</td><td>– must change how content is served</td></tr>
<tr><td>+ incremental deployment for scalability</td><td>+ scalability</td><td>– costs for scalable solution</td></tr>
</table>

Table 2.3: Where to adapt? Comparison of receiver-, proxy-, and sender-based adaptation.

potential that is inherent in the application. An additional drawback is that a system-based approach requires modifications to the operating system. This requirement can drastically hamper deployment, especially if necessary on the client-side.

System/Application. The collaborative effort between system and application (termed *application-aware* adaptation by Noble [130]) is attractive for the following reasons. The application can decide on the adaptation strategy, and thus make application-specific, quality-aware adaptation decisions. Knowledge of application resource needs and the advantages of a central authority for tracking end-system and network resource availability mentioned above complement one another ideally to allow for system-aware adaptation decisions.

These advantages come at the cost of requiring application and operating system changes, which may hinder deployment. Furthermore, since the application decides on behalf of the user how the quality is to be adapted to meet the network resource supply, the decisions may not exactly reflect the user's notion of quality.

Application. Application-centric (or *laissez-faire* [130]) adaptation allows for quality-aware decisions. The applications get the adaptation behavior they want. Again, knowledge about application resource needs helps to make informed adaptation decisions. Furthermore, no system support is required, which is a plus when it comes to deploying the application.

On the down side, being external to the system means that the application is less well-positioned to monitor end-system and network resources. (The effects of application concurrency are difficult to track.) As a consequence, the adaptation is likely to be less agile. Again, deciding on behalf of the user may not fully reflect user-intended adaptation behavior. Furthermore, each application must be programmed to support adaptivity. (A framework-based approach as proposed in this dissertation may leverage some of the development costs.)

	System	System/Application	Application	Application/User
Quality-awareness	− no quality tradeoff	+ application-specific, quality-aware decisions possible		− choices not meaningful
	− no user control, app./system decides on behalf of user			+ full control
System-awareness	+ central authority, system can enforce resource allocation		− no support for application concurrency	
	− unknown requirem.	+ application resource requirements known		
Dynamics	+ agility	± agility		− agility
Resource tracking	+ central authority for monitoring, accurate and efficient		− external to system: monitoring complicated, less accurate	
Deployment/ Development	+ application transparency	− application modifications necessary, complicates programming model		
	− requires OS modifications		+ no system support required	

Table 2.4: Who decides? Comparison of the different approaches.

Application/User. User-controlled adaptation differs from all the other approaches in that the user is in full control to make the adaptation decisions she intends and to get the quality she wants. The problems, however, are that adaptation choices which are offered to the user by the application may not be meaningful to the user, e.g., because they are often specified in terms of low-level system and network QoS parameters, such as bandwidth or delay. Unless the user has a clear understanding of the application resource requirements, it may be difficult for her to foresee the effects of the adaptation choices presented. A second concern is that with the user in the loop, application agility may suffer drastically.

Table 2.4 summarizes the discussion of the four alternatives.

Summary

The following paragraphs briefly review the related work and discuss whether or how the challenges identified in Chapter 1 are addressed.

Quality-awareness. No proxy-based solution has shown the potential of enabling quality-aware adaptation yet. Client-based solutions would require considerable amounts of coordination with the peer entities that perform the transcoding. Video applications often leave no choice in how they adapt, since most of them only implement one of the distillation-axes mentioned in Table 2.1, e.g., selective frame dropping [72], or hierarchical filtering [113, 152]. Solutions that have the operating system decide on how to adapt (e.g., [93]) cannot make a tradeoff for quality either. This leaves us with only one system, InfoPyramid [124], which takes an integral view at the bandwidth-quality tradeoff to be achieved by network-aware applications.

System-awareness. There are only a few network-aware applications that take the CPU resources needed for a bandwidth-quality tradeoff into account. Odyssey [132] and the study by Han [69] consider client CPU resources. The only study that takes a look at how the transcoding costs influence dynamic adaptation decisions is the one conducted by Han et al. [70]. In their

analytical framework for transcoding proxies, they use estimates for the transcoding costs and for the available bandwidth to determine whether transcoding would be able to speed up content delivery.

Application dynamics. The ability of the video applications to adapt to client heterogeneity and bandwidth fluctuations has been evaluated either by simulation [113, 152] or by live experiments [72]. With the exception of the Odyssey prototype most other systems either have not been evaluated at all or have not gone through more than an exemplary evaluation that shows that their approach may work (e.g., [93]). Noble et al. [132, 130] address two important questions. First, how agile is the application (or system) in the face of changing network bandwidth? Second, does adaptation provide any benefit to individual applications? Although seminal in the methodology used, the evaluation exhibits room for improvements. E.g., quality metrics are defined to establish that adaptation can provide benefits to the applications; however, these metrics are not used by the adaptation process itself and thus appear to represent fairly arbitrary choices. The evaluation does not try to establish which are the key factors that make adaptation work successfully (e.g., is agility the only relevant factor?).

Application-network coupling. Hardly any related work addressed the importance of application-network interaction. Although Noble et al. have shown that the Odyssey system (with its fixed strategy for bandwidth tracking) is agile enough (for the three applications considered), many questions remain unanswered. E.g., how much agility is actually needed to make adaptation perform reasonably? What impact on performance has to be witnessed if the system is less agile or if the bandwidth estimates are less accurate?

Ease of development/deployment. Proxy-based solutions can be deployed most easily, followed by sender-based approaches. Application-level adaptation is to be preferred to approaches requiring cooperation from the operating system, unless changes to the system have a very limited scope, e.g., only affect a single server. Issues relating to the development of network-aware applications will be treated in the following sections.

2.4 Reuse

As we have seen in the discussion above, network-aware applications that can cope with heterogeneity and fluctuating bandwidth are fairly complex software systems. Mastering complexity in software construction has been notoriously difficult. Once mastered successfully, one would like to reuse the software architectures, designs and source code (to speed up development of similar applications for instance). However, as experience shows, reuse does not simply happen, systems must be designed for reuse. This section first briefly mentions some common reuse techniques, then discusses related work where such techniques have been applied to enable reuse of (parts of) network-aware applications.

2.4.1 Reuse techniques

Various researchers and practitioners have embraced the paradigm of object-oriented design and implementation to achieve modularity and to increase software reuse [78, 155]. There are many techniques (or concepts) that foster software reuse: subroutine libraries, toolkits, design patterns [65], frameworks [86], or components [180] to name a few. We briefly describe those techniques that have been applied to the construction of network-aware applications, omitting subroutine libraries.

Toolkits. A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality [65]. Generic class libraries for basic data structures such as linked lists, stacks, queues, heaps, etc. provide an example of a toolkit [175]. The C++ I/O stream library is another example. Toolkits don't impose a particular design on the application; they provide building blocks upon which an application can be constructed (much like a subroutine library does). Therefore, toolkits emphasize *code reuse.*

Frameworks. A framework is an abstract design for a particular kind of application, and usually consists of a number of cooperating classes [86]. The abstract design provided by the framework can be customized to form a particular application. Customization is often achieved by creating application-specific subclasses of abstract classes from the framework. In contrast to toolkits, frameworks dictate the architecture of the applications derived from them [65] and define the flow of execution [181]. A framework captures the design decisions that are common to its application domain. Frameworks thus emphasize *design reuse* over code reuse. A distinguishing feature of this sort of reuse is that it leads to an inversion of control between application and the software on which it is based [181]: Applications call toolkits (or libraries). A framework calls the application, that is, the application-specific code. Usually, two types of frameworks are distinguished [86]: *white-box* and *black-box* frameworks. A white-box framework is a framework in which components are reused mostly by inheritance. Black-box frameworks achieve reuse of components by composition. An example of a white-box framework for the construction of an optimizing compiler is described by Adl-Tabatabai et al. [1]. The protocol framework implemented by Hüni et al. [76] is an example of a black-box framework.

Libraries and toolkits are often more widely applicable than frameworks and thus allow for more flexible reuse. Frameworks are usually very specific to a particular application domain, because they dictate the flow of control for applications derived from the framework. On the other hand, carefully crafted frameworks allow for much more powerful—since design-level—reuse and have therefore a higher potential to increase productivity. The applications derived from a framework have similar structures since they share the same flow of control. As a result, they are easier to maintain and appear more consistent to their users, that is, to application developers. The flip side of a framework-based approach to the development of software is the difficulty to design good frameworks and the often high learning efforts required to apply them.

2.4.2 Reuse in network-aware systems

Libraries

Odyssey. Odyssey [132] provides system support for network-aware applications and is implemented in the NetBSD operating system. Odyssey's functionality is provided by extensions to the operating system application programming interface (API). The API extensions are contained in a library that can be linked with the application. Each of the three applications built on top of the Odyssey API [130] has a different application structure, although at least the video player and web browser exhibit enough common traits that would allow for a more sophisticated type of reuse.

Toolkits

Software Feedback Toolkit. Cen [30] developed a toolkit for software feedback systems. This toolkit applies linear control systems theory directly to the creation of software feedback systems, which are inherently non-linear. The technique involves decomposing the entire range over which a system must adapt into smaller sub-ranges, within which a well-behaved, linear control system is valid. If the system crosses the boundary between two sub-ranges, the system provides a form of meta-adaptation that switches to a different control system that is valid for the new range. Within the linear portions, formal analysis can be used to prove certain properties of the system. Using the toolkit, adaptive systems can be built hierarchically using a number of basic feedback components, such as low-pass filters, hysteresis-based switches, multiplexors, and so on. The work of Cen demonstrates the applicability of the toolkit with examples from the realm of adaptive multimedia systems. In [31], Cen et al. describes an audio/video-player based on multiple smaller scale feedback systems: a packet-rate control based on packet-loss and latency feedback, a flow and congestion control system for Internet media streaming, and a control system used for audio/video synchronization.

MASH. Merging "best practices" from three multimedia toolkits (MIT's VuSystem [73], LBL and UCB's MBone tools vat [81] and vic [112], and Berkeley's continuous media toolkit [169]), McCanne et al. [110] describe a programming infrastructure that facilitates the construction of networked continuous media applications. The programming infrastructure, called MASH, promotes reuse at three levels. First, it describes a "system architecture" that models a continuous media stream as a collection of source, filter and sink objects. A media stream is generated by a source object and is piped through one or more filter objects before it reaches the sink object where it is consumed. Second, MASH is a toolkit containing a variety of source, filter and sink objects. Sources might be video capture devices, filters might be color space converters, compressors, packetizers, etc., and sink objects might be network transmission protocols (e.g., RLM [113]) or playout devices. Third, a programming model based on so-called "split objects" forms the core of the MASH toolkit. A split object is an abstract entity whose methods can be implemented either in C++ or in an object-oriented extension of the scripting language Tcl [135] called OTcl [196]. Method calls can be invoked from either side (C++ or OTcl) and are dispatched to the appropriate implementation. If efficiency is of outmost importance methods should be implemented in C++. If flexibility (e.g., to support rapid prototyping) is the

major concern, methods are preferably implemented in OTcl. The source, filter and sink objects mentioned above are all based on this programming model. By means of composition, these tools can be assembled to build applications or to provide higher levels of abstraction for the developer. McCanne et al. [110] claim that the split object approach at developing networked software promotes easy and flexible code reuse.

Frameworks

TACC. Infrastructural proxies must be scalable to large number of users and must be highly available. To provide a general solution that meets the challenges of scalability and availability, Fox et al. [61, 57] introduce a programming model for Internet services (TACC). TACC is based on **t**ransformation (distillation, filtering, format conversion, etc.), **a**ggregation (collecting and collating data from various sources, as search engines do), **c**aching (both original and transformed content), and **c**ustomization (maintenance of a per-user preferences database that allows transformation "workers" to tailor their output to the user's needs or device characteristics). The cluster-based TACC server architecture described in [61] serves as a framework for building adaptive network services. The developers of new services can use this framework as an off-the-shelf solution to scalability, high availability, and fault tolerance, and can thus focus on the content of the service being developed. The developer of a proxy that must support transcoding for a new media type, for example, must merely devise an appropriate transcoding module (or worker) and register it with the TACC-framework. TransSend [58, 60], a Web accelerator using dynamic distillation, is a sample instantiation of the TACC-framework.

Resuming the discussion of toolkits and frameworks, we find the pros of both approaches confirmed by the related work cited above. The two toolkits (as well as the Odyssey library) allow for very flexible reuse. Many diverse applications can benefit from reuse. On the other hand, using the TACC-framework relieves the developer of a new application of the burden to bother with how to structure it. Considering the challenges for network-aware content delivery identified (Chapter 1), we expect adaptive applications addressing all the issues of quality- and system-awareness to be highly dynamic and complex software systems. The core adaptation process must take many parameters into account (e.g., network and end-system resource availability, application resource demands, quality expectations, etc.) to make suitable adaptation decisions and will therefore exhibit a highly non-trivial control flow. Compared to toolkit-based software development, a framework-based approach is better suited to shield developers from both the structural and run-time complexity of a system addressing the aforementioned challenges, mainly because it allows for reuse of the control flow, that is, the entire adaptation process.

2.5 Summary

This chapter discusses three approaches that could be used to provide a service model for networked applications that allows to achieve predictable response times while maximizing the "value" of the data delivered to the user. First, greedy—since unresponsive, and non-congestion-controlled—transmission of data encoded according to the principles of application

level framing can have a disastrous impact on Internet stability and is therefore not a viable option for such applications. (Furthermore, greedy data transmission is not a viable option because routers will be able to isolate and "punish" such non-conforming applications in the near future [53].) Second, exploiting the reservation-capabilities of an integrated services packet network is an attractive approach that can simplify application development. However, reservations do not solve all the problems, as an application may be required to adjust its resource demands to meet a granted reservation. Therefore, we conclude that adaptation—the third approach—is an attractive and often necessary means to achieve the objectives of timely content delivery.

The discussion of related research on network-aware content delivery reveals that the five challenges listed in Chapter 1 (quality-awareness, system-awareness, application dynamics, application-network integration, and ease of deployment) have hardly been addressed so far. To the best of our knowledge, there is no prior work which provides an integral approach that tackles all the challenges. Some of the applications or systems described by previous work have covered individual aspects. The classification of the different approaches to network-aware content delivery prepares the ground for the design decisions explained in the next chapter.

Based on the review of commonly used techniques for software reuse and the discussion of related work that applies these techniques to enable reuse of (parts of) network-aware applications, we conclude that a framework-based approach to the development of network-aware applications is best suited to meet the challenges put forth.

Chapter 3

Framework concepts

Application frameworks capture the design decisions that are common to their application domain. This chapter first spans the application domain (Section 3.1) and details the service model for network-aware applications that address the challenges listed in Chapter 1 (Sections 3.2 and 3.3). By drawing on the results of the discussion of related work in Chapter 2, this chapter then describes implications of the service model and presents the concepts and design decisions that form the basis of our framework-based approach to the development of network-aware applications (Sections 3.4 and 3.5). The design decisions to be made include the choice of the component of a network-aware application that is responsible for the adaptation decisions (sender, receiver, or intermediary network nodes), and the structure of the framework.

3.1 Application domain

The application domain of our framework is characterized as follows: we focus on *client-server* applications that allow users (clients) to request services from a remote site (server). The remote site responds with the delivery of data that satisfies the user's request. We assume that the response from the server contains a set of objects (text, images, video or audio sequences, byte code, etc.), which are retrieved from secondary storage (e.g., by a file server) or computed on-the-fly (e.g., by a search engine). In such applications, the response usually has a considerably larger volume than the request and dominates the transmission costs. Hence, the response from the server includes a *bulk transfer* across a *best-effort* network. As a direct consequence of the discussion in Section 2.1, we assume that this bulk data is transferred from the server to the client using a *congestion-aware* transport protocol such as TCP [176]. In the following, we sometimes refer to server and client as sender and receiver (of the bulk-transfer).

This definition of the application domain is widely applicable and comprises a large and important class of client-server applications currently deployed in the Internet. Many of these applications can benefit from an enhanced service model that provides predictable response times (see below). The discussion of concrete applications is deferred to Chapter 5.

Note that this definition of the application domain is restrictive and limits the applicability (and reuse potential) of the framework. E.g., the definition excludes multi-party applications that rely on multicast content dissemination and it limits the direct applicability of our framework to continuous media applications transferring delay-sensitive data. These restrictions are

introduced because there already exists a significant body of related research that covers these two areas (see Section 5.2.4 for a discussion).

3.2 Service model

Since the latency of congestion-controlled bulk transfers in a best-effort network may be highly unpredictable, the *service* provided by such client-server applications can be enhanced by allowing the user to specify how long it is willing to wait for the delivery of the requested data. The application should then strive for a predictable response time and should try to fully utilize the time frame allotted (to maximize the quality of the data delivered). The minimal set of (user) interactions with such an application (and hence the service model to be supported by the framework) can be described as follows.

User-requests must comprise three parts: (i) a set of objects to be retrieved (or computed), (ii) quality restrictions on the objects, and (iii) a limit T on the response time. The set of objects can be either explicitly or implicitly specified by the user. For example, if the user requests a (compound) document that consists of multiple objects, the in-lined objects are implicitly requested. The quality restrictions characterize the minimum quality of the objects delivered that is tolerable and the maximum quality that is beneficial for the user. The upper bound on quality may be imposed by the requester's processing or display capabilities. Thus, the quality restrictions may also be implicitly specified, e.g., by the type of the client device. Lower bounds specify the minimal quality an object may have and still be useful to the user. Note, that (some of) these bounds may also be unspecified, in which case application defaults apply.

The goal of the framework, and hence the *service* provided by applications built thereon, is to deliver the requested objects to the user within the time frame allotted. For this purpose, the application may adapt, that is reduce, the quality of the objects transferred to the client. The range for quality adaptation is bounded by the quality restrictions on the objects. If the objects must be adapted to meet the goal on the response time, the adaptation must happen such that the overall quality of the entire response is maximized (the term "overall quality" is defined in the following section).

3.3 Quality

Central to the service model is the definition of the term *quality*. The notion of quality is clearly user- and application-specific. Only the application (developer) knows what quality is. So the central issue is that we must find a software structure that allows the application developer to specify what quality means in the context of a specific application. In the context of the service model outlined, two general aspects can be identified that characterize the overall quality of a response: (i) the quality of an individual object, and (ii) the importance of an object within its context, that is, the relevance of the object.

Object quality. How to grasp the concept of object quality? To prepare the grounds for a quality-aware decision making by network-aware applications, we must be able to compare the quality of two objects (or two versions of the same object) and to quantify the difference in

quality. The term quality should capture the data's "usefulness" or "value" to an end-user. The quality of an object may have many datatype-specific attributes. E.g., the quality of an image could depend on the resolution, color depth, level of compression, and so on. As a consequence, the value of an object may be effected by a number of datatype-specific factors whose contributions to the value depend on the application context. To be useful in a general setting of a framework for network-aware applications the abstractions for object quality must be flexible enough to be applicable to many data types and application contexts. We use *utility functions* to quantify an object's value (or utility) to the end-user[1]. Utility functions are (multivariate) functions that map the potentially multiple dimensions of quality into a single real number in the range $[0, 1]$. A utility of 0 means that an object has no value to the user. A utility of 1 implies maximal value. Utility functions provide a flexible abstraction to characterize quality. They allow to quantify object quality with a single number, which can be easily used by generic, application-independent adaptation strategies. In step with the registration of the datatypes handled, the application can define appropriate utility functions for each of the data-types supported[2].

Importance of an object. How to characterize the importance of an object within its context? Again, this is highly application- and user-dependent. While the definition of object quality typically depends on the data-type, the definition of importance is very content- and context-dependent. (E.g., advertisements in a Web page may have low relevance, while other images may be more important to a user.) Again, for reasons of flexibility, we characterize the importance of an object within its context by means of a single number, which we will call the *relevance score*.

A definition for the overall quality of a response follows quite naturally using utility functions to characterize the quality of individual objects and relevance scores to define the importance of an object relative to its context. We define the *overall quality* of a response as the weighted sum of the individual object qualities, that is, as the weighted sum of the utility values for each of the objects. The weights for the objects are given by the relevance scores.

3.4 Implications of the service model

The service model calls for two types of adaptation: adaptation to client heterogeneity and adaptation to network variability.

First, the bounds on the minimum quality tolerable by and the maximum quality beneficial to the client require the application to adapt to client capabilities. Otherwise, precious network resources may be wasted. If an object is delivered in a quality lower than the minimum quality,

[1]Utility functions are a widely used concept to quantify the value or outcome of a complex situation, decision, etc. Utility functions have been used by other researchers in the network community, e.g., by Breslau et al. [24] (see Section 2.2). Utility functions are very popular in the economic literature on the topic of risk management and decision making. Utility theory [120] provides the mathematical foundations for utility-based decision making.

[2]An alternative approach to characterize the value of an object is to use an application-independent, e.g., information-theoretic measure, of the content delivered. Mohan et al. [124] follow such an approach in their InfoPyramid system (see Section 2.3.3). Their approach is based on the rate-distortion theory by Shannon [164]. While elegant because of their application-independence, such approaches at quantifying the value of content may not always correspond with user perception [124].

utility for the user is zero and the object is transmitted in vain. Likewise, if the maximum quality is exceeded (e.g., if an image is sent in a higher resolution than supported by the client's display), bandwidth is wasted without a gain in utility.

Second, the goal of maximizing quality dictates that the application must account for both network heterogeneity *and* bandwidth volatility. Adapting to bandwidth fluctuations means that the application need not only dynamically adapt to decreasing bandwidth (by reducing quality), but should also try to opportunistically exploit extra bandwidth (by "revoking" previous adaptation decisions) to deliver as many high quality objects as possible within time T.

As a consequence, network-aware applications adhering to the service model above must address the following two questions: (i) how to find out about (dynamic changes in) the bandwidth available on the path from the sender to the receiver, and (ii) how to adapt the delivery process (to such dynamic changes) such that the objectives of the service model are met. For the sake of brevity we refer to these two tasks as (i) "resource discovery", and (ii) "adaptation" in this chapter.

This section discusses implications of the service model for the design of network-aware applications. The design decisions are justified by drawing on the results of the previous chapter.

According to our classification of related work (Section 2.3), there are two important design decisions to be made for network-aware applications. First, which party in a networked application is responsible for (i.e. initiates) the adaptation: the sender, the receiver, or intermediary nodes in the network (e.g., proxies)? This question is addressed in Section 3.4.2. Second, which entity in the adaptive application is responsible for resource discovery and making the adaptation decisions: the operating system, the application, both, or the user? Section 3.4.3 answers this question.

Furthermore, design decisions must be made as to how to structure a network-aware application that provides a service such as defined in Section 3.2. There are three issues pertaining to the problem of structuring such network-aware applications: how to layer the functionality (adaptation and resource discovery) within the application, how to divide the functionality of the adaptation process between application and framework, and finally how to structure the adaptation process itself. These issues are treated in Sections 3.4.4, 3.4.5 and 3.5, respectively.

3.4.1 Model-based adaptation

Before we can turn to resolving the questions listed above, we must first study how the requirements for quality-aware adaptation that are laid out in Sections 3.2 and 3.3 can be met. In other words, we must study how such network-aware applications must adapt to meet their goals on response time and quality.

Steenkiste [171] identifies three generic adaptation models for network-aware applications. We briefly review the three models.

Performance-based adaptation. The application monitors its performance (e.g., its throughput) and controls adaptation based on these observations. The control parameters (e.g., the sending rate) are typically adjusted incrementally, because there is not enough information available to calculate the parameters explicitly. By monitoring the application performance (e.g., by observing how many packets are dropped), network performance is obtained only implicitly. The main advantage of performance-based adaptation is its

simplicity (for applications that have a simple definition of performance). The adaptation is robust, as it does not depend on accurate information about resource supply and demand. The drawbacks are that such a scheme is purely reactive and that it allows only for "one-way" adaptation. Such a scheme is reactive, because it merely reacts to service degradations, but does not try to anticipate them. The adaptation is "one-way", because degradation in network resource availability can be observed (and reacted to), but probing is necessary to learn about a sudden abundance of network resources. Performance-based adaptation is employed in TCP [176] and some of the video applications mentioned in Section 2.3 [113, 152].

Feature-based adaptation. The application monitors some feature of the application and uses that information to adapt. A "good" feature correlates with application performance. This type of adaptation can be viewed as a generalization of performance-based adaptation (performance is a feature of an application). However, careful choice of the feature helps eliminate some of the drawbacks of performance-based adaptation: feature-based adaptation can be proactive and symmetric. That is, changes in resource availability may be anticipated, and there is no need to probe for resource availability. TCP Vegas [22] and the video application developed by Hemy et al. [72] serve as examples here.

Model-based adaptation. In contrast to the alternatives above, the application has a model of its performance as a function of the various parameters characterizing its run-time environment,e.g., network bandwidth, transcoding costs, and so on. Given information about the run-time environment, the application uses the model to select the settings for the control parameters that will give the best performance. The advantage of model-based adaptation is that it can potentially quickly zoom in on the right control parameter values. A disadvantage could be its robustness, as the adaptation may be sensitive to the accuracy of network status information and the correctness of the model. Building an accurate model may be difficult for complex applications. On the other hand, complex applications cannot benefit from the simplicity of performance- or feature-based adaptation either, because it is difficult to find an appropriate feature that correlates with application performance. A sample application performing model-based adaptation is described by Siegell et al. [168].

The question is, which of these adaptation models is best suited to the type of network-aware applications specified by the service model? The service model dictates that the network-aware application should strive for predictable response times and maximize quality of the objects delivered within the user-specified time limit. Response time is an end-to-end metric. As such it covers all aspects of the networked application which may effect the response time. E.g., the response time not only covers the time required to transfer a set of objects (images, text, ...) across the network, but also includes the time needed to compute, transcode, or retrieve the objects. Thus, a first consequence of the service model is that the adaptation must be *system-aware* (as well as network-aware) and must take at least the CPU resources into account that are needed to make the quality-time tradeoff.

The service model further implies that application performance depends on two issues, the application's ability to meet a time limit and its ability to maximize quality. Performance- and feature-based adaptation are ill-suited to fulfill the goals of the service model for the following

reasons. First, the complexity of the performance definition makes it difficult to find an appropriate application feature that correlates with performance. Second, even if such a feature existed, it would be application-specific and thus hardly useful in the context of an application framework. And most importantly, implicit knowledge about application performance (as obtained by observing an application feature) does not suffice to achieve an explicitly quantified performance goal (that is, to meet a user-specified time limit). As a consequence, we employ model-based adaptation.

3.4.2 Sender-initiated adaptation

We now address the question which party in a networked application is best suited to initiate adaptation: the sender, the receiver, or a proxy? Reviewing the discussion of the three approaches in Section 2.3.4, we find that a proxy-based solution exhibits severe deficiencies when it comes to quality-aware adaptation. Limited to process one object at a time and lacking information about the context of a particular object, a proxy can often only make "local" decisions. Therefore, proxies are ill-suited to perform an integral quality-time tradeoff. Moreover, for any proxy-based solution the issue of where to place the proxy must be resolved. As automatic placement and location of proxies remain an active area of research [6], proxies must typically be configured statically.

There are three reasons that favor sender-initiated adaptation over receiver-initiated adaptation. First, a sender-based scheme leaves the content provider in control about how content should appear at the client (if he should decide to override the preferences stated by the user). Second, in contrast to sender-initiated adaptation, receiver-initiated adaptation would call for elaborate coordination with the peer (the server), which performs the necessary transcoding. Third, since the response time metric calls for system-awareness, the (CPU) costs both for the transcoding at the server and for the presentation at the client must be considered for adaptation decisions. CPU resource availability at the client is likely to be more easily predictable than the resource availability at the server [41]. As a result, the sender-initiated approach can more reliably make tradeoffs that incorporate client capabilities than a receiver-initiated scheme, which would have to take server resource availability into account.

3.4.3 Application-level adaptation

As far as the question about the entity that controls adaptation is concerned, we opt for application-level adaptation. This decision is driven by the requirement that the network-aware applications derived from our framework should be easily deployable, as well as by the objective to find a compromise between quality-awareness and agility. First, the requirement for ease of deployment precludes approaches that involve the (operating) system in the adaptation process and hence require modifications to the system. Second, quality-awareness would ideally be achieved by user-level adaptation. However, a user-based approach would come at the cost of agility, that is, the application's ability to react swiftly to changes in the network (see Section 2.3.4). Therefore, application-level adaptation represents a reasonable tradeoff between these two conflicting issues.

3.4.4 Application layering

As we have settled on sender-initiated, application-level adaptation, we are now left with the question of how to layer the functionality within the sending application. As stated above, network-aware applications must tackle two problems: (i) network resource discovery, and (ii) adaptation. We separate the treatment (and the implementation) of these two issues for the following reasons. First, although adaptation depends on the information about network status, the two issues can be considered to be orthogonal. Second, both on-line network resource discovery and dynamic network adaptation are complex tasks. Thus, a split of mechanisms for adaptation and mechanisms for network resource tracking helps reduce software complexity. Third and most importantly, solutions to the two problems exhibit considerably different reuse potentials. The adaptation mechanisms are tightly coupled with the service model and are therefore useful only for applications from the application domain sketched above. On the other hand, mechanisms for network resource discovery/tracking are useful in a much broader scope. As a consequence, the issue (ii)—how to adapt the delivery process—is captured by a framework which is described and evaluated in Chapters 4–6. Solutions to the issue (i)—how to find out about dynamic changes in available bandwidth—are provided in a toolkit, which is detailed in Chapters 7–9.

3.4.5 Framework versus application functionality

The question that we address next is: what parts of the functionality of network-aware adaptation can be captured in a reusable framework, and what remains to be done by applications derived from the framework?

The answer to this question follows quite naturally from the definition of the service model in Section 3.2. Clearly, the application (developer) must specify the kind of objects (i.e., the data types) that can be requested/handled. The application must also define its notion of quality, that is, how to interpret and quantify the quality of a response (Section 3.3). And finally, only the application developer knows which options for quality reductions should be supported (and these options typically are different for each data type).

The framework must then provide application-independent adaptation mechanisms that aim at providing predictable response times while maximizing the quality of the objects delivered. The adaptation mechanisms captured in the framework are detailed in the following chapters.

Note that client functionality (user interaction, caching, server selection, etc.) could also be divided into application-dependent and application-independent components. However, as our focus lies on adaptation mechanisms and their dynamics, we do not consider client issues in the design of our framework and leave it to the application developer to provide the corresponding client functionality. We also leave it to the application developer to provide the user with additional flexibility to interact with the framework, e.g., to define and change the notion of quality at run-time as proposed in [116, 45].

3.4.6 Application structure

The definition of the service model and our findings from the discussion of related work entail a number of implications for structuring network-aware applications which we briefly summarize

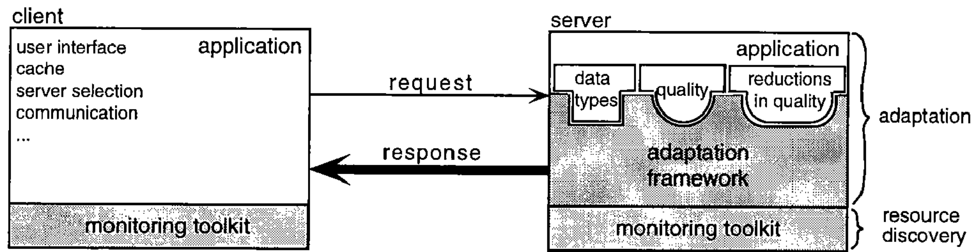


Figure 3.1: Application structure for sender-initiated, application-level adaptation.

here. Figure 3.1 schematically depicts the chosen structuring of network-aware applications adhering to the service model defined in Section 3.2. The figure illustrates and summarizes the four design decisions adopted in this section: first, we concluded that for reasons of quality- and system-awareness adaptation should be sender-initiated. Second, application-level adaptation facilitates deployment and represents a good tradeoff between agility and user control. Third, complexity arguments and the reuse potential lead us to separate adaptation mechanisms from mechanisms for network resource discovery and monitoring. Finally, the adaptation mechanisms are captured in the framework to be detailed in the following. Three aspects of application-specific functionality are factored out of the framework: the data types handled by the application, the algorithms applicable to achieve quality reductions for a particular type, and the application's notion of quality.

3.5 Framework structure: Feedback control loop

The framework essentially captures the adaptation process, thus, to come up with a design for our framework, we must first reason about suitable structures for the adaptation process. A useful structure for network-aware applications using request-response type communication is a software *feedback control loop* [30, 17], where the time left for the response—initially set to T—constitutes the command variable of the closed-loop control. The feedback driving the sender adaptation comprises information about the currently available bandwidth as obtained by mechanisms discussed in Chapter 7.

We model sender-initiated adaptation in a closed-loop control system with the three phases *adapt* (P_{adapt}), *prepare* (P_{prep}), and *transmit* (P_{trans}), as depicted in Figure 3.2. The three phases share the list of requested but not yet transmitted objects. P_{adapt} is responsible for obtaining information about the available bandwidth, determining whether the amount of data to transmit must be reduced or whether it may be increased. In case adaptation is needed, P_{adapt} must decide which objects to adapt, and which transformations to apply. The term "transformation" refers here to any activity including transcoding, conversion, or computation. For each object, these decisions are recorded in a so-called *quality state* that is attributed with each of the objects in the request list. Once a (final) decision on the quality of an object to be delivered has been made, P_{prep} must carry out the transformation of the object to the quality that has been assigned by P_{adapt} and which is reflected by the object's quality state. P_{trans} delivers completely prepared objects to the client. The three phases are executed repeatedly. To keep the overhead incurred by transformations as small as possible, P_{adapt} does not invoke the transformations directly (after an adaptation decision has been made), but defers their execution to forthcoming phases P_{prep} to

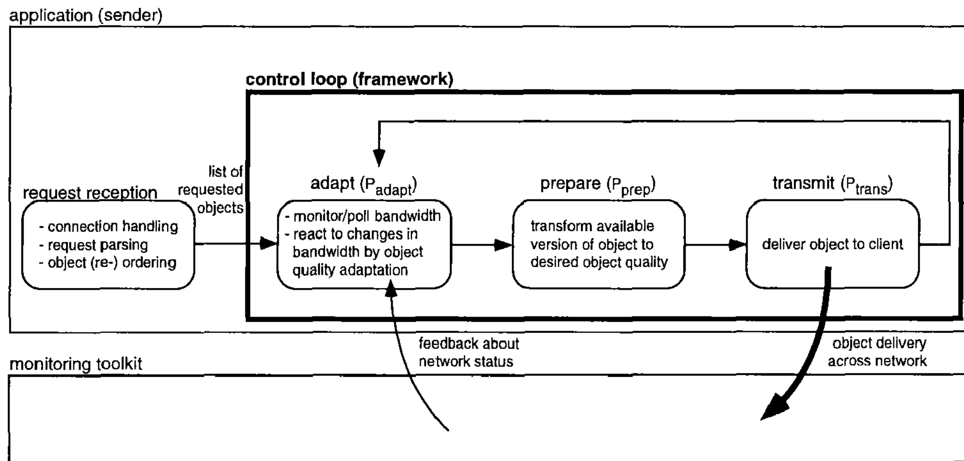


Figure 3.2: The control loop, which is deployed at the sender, consists of three phases *adapt* (P_{adapt}), *prepare* (P_{prep}), and *transmit* (P_{trans}).

allow for "last-minute" adaptation. While P_{adapt} may need to change the quality state of several objects at the same time, P_{prep} makes only one object ready for transmission at a time, that is, in one iteration of the control loop.

An important concern that must be considered when designing a feedback control system is the issue of *potential instabilities*. A control system is said to be instable if its response to an impulse (e.g., a change in the network environment) does not lead to a constant behavior as time approaches infinity, that is, if the system's response variable oscillates indefinitely. Section 4.5 describes how the concern of potential instabilities in the adaptation behavior is respected in our framework. The evaluation of a sample application derived from the framework shows that sender-initiated, model-based adaptation is robust with respect to most of the parameters that influence adaptation decisions (see Chapter 6). In other words, the evaluation shows that adaptation produces stable results regardless of how the factors influencing adaptation decisions are varied.

3.6 Summary

The chapter sketches the application domain and describes the service model for the type of applications supported by our framework-based approach to the development of network-aware applications. The definition of the application domain is widely applicable and comprises a large and important class of client-server applications currently deployed in the Internet. The service model dictates that applications adhering to the service model must attempt to deliver the requested data within a user-specified time frame and must try to maximize the quality of the content delivered. The chapter introduces abstractions to characterize the *quality* of a response. The definition of quality takes the quality of individual objects as well as the relative importance of the objects within a request into account.

Merging the findings from related work with the implications of the service model, we arrived at the following design decisions, which take the requirements stated in Chapter 1 into consideration. First, adaptation decisions should be made at the application-level to ease de-

ployment of network-aware applications. Second, the sender should be in charge to make the adaptation decisions. Sender-initiated adaptation supports an integral approach to service quality. Third, model-based adaptation is well suited to meet a quantitative performance target, such as a predictable response time, because it accounts both for network and end-system resource availability when making adaptation decisions.

To reduce complexity of the software system, we separate the adaptation functionality from the mechanisms to discover and monitor bandwidth availability. The adaptation mechanisms are captured in the framework, while the mechanisms for network resource discovery are provided by a toolkit. Finally, the chapter proposes to structure the adaptation mechanisms as a software feedback control loop consisting of three phases adapt, prepare and transmit.

Chapter 4

Feedback loop and adaptation

4.1 Overview

As stated in the previous sections, the goal of a network-aware sender is to meet a user-specified bound on the delivery time by adapting the quality of the objects delivered to the available network capacity. Thereby, the objective of the adaptation process must be to utilize the available resources as efficiently as possible and therefore to maximize the user-perceived quality within the bounds (time, bandwidth, and boundary conditions on quality) given.

This chapter describes the implementation of the application framework that provides a solution to one of the two problems network-aware applications are faced with, the problem of finding out when and how to adapt. The discussion of these two issues proceeds top-down: Sections 4.2 and 4.3 introduce the basic ideas of when and how to adapt, respectively. Section 4.4 describes the response time model which is used to determine when to adapt, and Section 4.5 details the decision-making process and answers the question how to adapt. While Sections 4.2–4.5 describe the framework dynamics and point out where and how application-specific information can (and must) be factored out of the software control system to provide a reusable framework, Section 4.6 describes the static structure of the framework. Sections 4.7 and 4.8 discuss adaptation inherent problems such as start-up and agility. Section 4.9 briefly touches additional adaptation-related issues and summarizes the chapter. Abbreviations that are frequently used in the following chapters are listed in Table 4.1.

The adapt phase (P_{adapt}) is the key phase in the framework. It is repeatedly invoked and is responsible for the following tasks. First, it must obtain feedback about network status. Second, based on this information it must decide whether quality adaptation of the objects to be delivered is required or not. And third, if adaptation is required, P_{adapt} must decide how to adapt. The first two tasks are addressed in the following section. The third task is detailed in Section 4.3.

4.2 When to adapt?

There are two alternatives to obtain feedback about "relevant" changes in network service quality. First, P_{adapt} can poll information about network status and then decide about the relevance of a change in bandwidth. Second, P_{adapt} can determine the tolerance window for network resource availability (within which no adaptation is required) and register with the monitor layer

P_{adapt}	adapt phase
P_{prep}	prepare phase
P_{trans}	transmit phase
T	user specified time limit for response delivery
t_{left}	time left to deliver response, initialized to T
c_{prep}	CPU resources needed to prepare objects for transmission
t_{prep}	time needed to prepare objects
t_{trans}	time needed to transmit objects (given $bw(t)$)
t_{needed}	time needed to deliver response (given t_{trans}, t_{prep})
t_{diff}	error variable of control loop ($t_{needed} - t_{left}$)
d_{left}	data remaining to be transmitted
$bw(t)$	estimate of bandwidth available in $[now, now+t]$
$load(t)$	estimate of system load in $[now, now+t]$

Table 4.1: Abbreviations used in this chapter.

for asynchronous notification if service quality strays out of the tolerance window. (The second approach has been taken by Noble et al. [132], for example.) Whether a change is relevant or not may depend on a number of (interdependent) factors in addition to network resource availability (see Section 4.4). Since the factors may be interdependent, it is difficult to find meaningful tolerance windows for each of the factors individually. Thus, we are pursuing a polling-based approach for the sake of simplicity. The mechanisms employed for network resource discovery are discussed in Chapters 7–9.

Two issues must be addressed to decide whether adaptation is required or not. First, what are the information requirements for the decision? Second, which changes in the application's run-time environment are significant enough to call for adaptation?

Since the response time is the primary "optimization" goal, P_{adapt} must try to estimate how much time it would take to transfer the (remaining) objects in the request list to the client if each of the objects is shipped "as is". For the purpose of estimating the time needed (t_{needed}) for the transfer, P_{adapt} must establish the amount of data remaining to be transmitted (d_{left}), and must have an estimate on how much bandwidth *will* be available on the path from the server to the client. We assume that P_{adapt} can request (from lower layers) estimates, or rather predictions, $bw(t)$ of the bandwidth available in the time interval between now and time t in the future[1]. d_{left} is the sum of the object sizes d_i of all the objects i which have not yet been delivered. For objects i that are retrieved from secondary storage, d_i is readily available. The size of objects that must be computed on-the-fly, or that must be transcoded (e.g., to meet client capabilities) may not be as easily obtainable. For such objects, d_i must be computed by translating the application-level quality metrics (e.g., resolution, color depth, etc. for image objects) to the storage space required to hold an object of equivalent quality. This translation is application-specific and must therefore be factored out of the framework. The application must provide a translation function $data : Quality \rightarrow \mathbb{N}$ for each class of objects supported.

Given d_{left} and a function $bw(t)$, we can compute t_{needed} by integrating (i.e. by summing up

[1]Inaccurate predictions may lead to suboptimal adaptation decisions, however, the repetitive nature of the control loop often allows to take corrective action as discussed at the end of Section 4.2.

piecewise continuous parts of) the function $bw(t)$ over time t until an area (i.e. data volume) is covered which matches d_{left}. t_{needed} represents the time needed to transfer d_{left} given bandwidth $bw(t)$. In a best-effort network the available bandwidth can hardly be predicted at the minute level of detail suggested by the function $bw(t)$. Thus, approximations are required. We assume that P_{adapt} obtains estimates of the *average* available bandwidth over a time frame $[now, now+t]$ (referred to by $bw(t)$)[2]. If t_{left} exceeds the time for which the predictor can reliably predict $bw(t)$, we make the simplifying assumption that the bandwidth is constant beyond that point.

Corrective action must be taken if t_{needed} and the time left to deliver the response, t_{left}, differ significantly. Significance depends on a number of factors, among others the size of the objects and the granularity of the quality adaptation possible. If the requested objects are large and if the granularity of adaptation is coarse, that is, if there are only a limited number of transcoding options, then the smallest delta in quality and hence data volume that can be achieved by adaptation is quite large. Thus, t_{left} and t_{needed} must differ substantially so that corrective action can be taken. The converse is true if objects are small and/or adaptation granularity is fine. For a more detailed discussion of *agility* aspects, see Section 4.8. To simplify the concept of "significance" we use an application-specified threshold ε to trigger adaptation (Section 4.5).

Note that P_{adapt} needs to estimate bandwidth $bw(t)$ only for the time needed to prepare and transmit the next object to be able to satisfy the user's request within the time limit—the reason is that the control loop gets an opportunity to take corrective action during the next iteration of P_{adapt}, if required. In case there are no such estimates, the bandwidth estimates are inaccurate, or if the application is not *agile* enough, that is, if transmission of the next object takes longer than the system can reliably predict $bw(t)$, the situation is slightly more complicated. Either the control loop gets a chance to take corrective action (because the time limit did not expire), or the data cannot be sent in the allotted time. In the latter case, the application must be able to deal with the breakdown of the service model (Section 4.9).

4.3 How to adapt?

The goal of P_{adapt} is to bring t_{needed} in line with t_{left} by either reducing or increasing the quality of the objects remaining to be delivered[3]; these actions thereby reduce or increase d_{left}. The following questions must be considered while the sender tries to compensate for the difference $t_{diff} = t_{left} - t_{needed}$ by adapting the quality of the data awaiting delivery:

Victim choice. Which object(s) should be chosen for adaptation?

Quality distribution. How should the amount of quality adaptation be distributed among the chosen objects? How to find the amount of quality adaptation needed given the volume of data adaptation required ($d_{diff} = t_{diff} \cdot bw(t)$)?

Algorithm selection. Which transformations should be used to accomplish a desired adaptation?

[2] In addition, some bandwidth predictors may provide confidence intervals $[bw_{min}(t), bw_{max}(t)]$ for the average bandwidth. Section 4.9 discusses how such information can be used to make more robust adaptation decisions.

[3] Recall, that P_{adapt} works on a so-called quality state defined for each object in the request list. *Increasing* the quality of an object means that previously assigned quality reductions, reflected in the current quality state, are undone.

These questions are application-specific, interdependent, and often ambiguous. They are application-specific, e.g., because the application defines the transformations applicable; they are interdependent, e.g., because the victim choice is influenced by the quality adaptations that are possible. They are ambiguous in the sense that there are often several alternatives which can achieve the quality-time tradeoff striven for. The service model dictates that the main objective for the decision making must be to maximize the overall quality when reducing $|t_{diff}|$. This criterion helps to disambiguate different alternatives that are all able to meet the time limit.

Victim choice. If quality reduction is required, objects should be chosen that have a small impact on overall quality (but ideally achieve a high data reduction). Conversely, in case expansion is needed, objects should be selected that achieve a high gain in overall quality (compared to the increase in data incurred). The abstractions provided by the definition of quality (Section 3.3) allow to tailor the victim choice to the application's specific needs, e.g., they let the application select the objects that are the least relevant; choose the objects with the smallest contribution to overall quality; or even pick objects randomly.

Quality distribution. Given a set of victims to be reduced (or expanded), ideally the individual objects are reduced in quality inversely proportional to their contribution to the overall quality of the response. The problem is that two objectives at different levels must be satisfied. On one hand, a balance of quality reduction must be achieved according to the relative importance of the victims. On the other hand, a certain amount of data reduction must be accomplished to match the available bandwidth and to meet the time limit. The main problem is that the mapping from network to application quality measures is generally ambiguous (in contrast to the application-to-network mapping performed to compute t_{needed}). E.g., to effect a size reduction of a factor α, images may typically be either scaled down by a factor $\sqrt{\alpha}$, or reduced in color depth by a factor α, compressed with an appropriate compression ratio that achieves the same reduction, or transcoded by a suitable combination of the three transformations. Due to this ambiguity and due to the difficulty of specifying quality in an application-independent manner (see Section 3.3) there is no generic inverse function $data^{-1} : \mathbb{N} \rightarrow Quality$ that would allow to *efficiently* carry out the quality distribution intended. A straightforward solution that simply computes and compares the data and quality reduction for all the algorithms applicable would be highly inefficient (Section 4.5.1).

Algorithm selection. The choice of the (transformation) algorithm to accomplish a given quality adaptation (or to produce an object at a given level of quality) is closely related to the issue of how much quality adaptation is required for each victim. There is usually an application-dependent choice as indicated above. The framework requires that the application specifies a list of transformation algorithms for each type of objects supported. Each algorithm must provide a list of parameter values applicable. In addition, the application must provide functions that help the adaptation process estimate the data and quality reduction potential of an algorithm on a per-object basis (see Sections 4.6 and 5.1.3).

The problem of making adaptation decisions is further complicated since (i) the adaptation potential of an object (limited by the boundary conditions on min/max quality) must be taken

into account, and (ii) the transformations applied on the objects consume host resources and time. Therefore, the transformations indirectly influence t_{needed}. The issues related to decision making are discussed in Section 4.5; issues related to (ii) are addressed in the following section.

4.4 Modeling response time

As indicated in Section 3.4.1, response time is an end-to-end metric and as such covers all aspects of a network-aware application which may effect the response time. That section concluded that adaptation must be system-aware and that model-based adaptation is best suited to meet the explicitly defined performance goal of a time limit. This section describes the performance model that lies at the heart of the framework's control loop.

4.4.1 System-awareness

There are a number of factors that effect the response time of network-aware content delivery. First and foremost, the available bandwidth determines the transmission time (Section 4.2). Second, quality adaptations, e.g., by means of transformations, consume processing power at the sender and may take a non-negligible time to be completed. Third, decision making during the adapt phases does not come for free either. Furthermore, presentation conversion (e.g., decoding) of the data delivered consumes CPU resources at the client.

Our performance model, which drives the adaptation process, must account for all the relevant factors for the following reasons: on one hand, quality reductions may result in the desired reduction of transmission time. On the other hand, the adaptation overhead implies higher CPU costs. Obviously, situations must be avoided where a reduction of object quality in an attempt to reduce the error variable $|t_{diff}|$ to zero incurs overheads, that is, prepare and adapt costs, $t_{prep} + t_{adapt}$, that are higher than the gain in transmission time (i.e. $t_{prep} + t_{adapt} > t_{diff}$).

We exclude the decision making overheads (t_{adapt}) and the costs for presentation conversion at the client from our performance model for the following reasons. For the sake of simplicity we assume t_{adapt} to be small enough to be negligible compared to t_{prep} and the transmission delays (this assumption is validated in Chapter 6). If this were not the case, that is, if t_{adapt} consumed substantial amounts of time, adaptation would be useless. Client-side overheads, e.g., for presentation conversion, are non-negligible, especially for resource poor mobile devices as shown by Han [69]. These costs can be excluded from the model, because they can either be entirely avoided or accounted for by making proper use of the flexibility provided by the service model (see Section 4.9).

As a consequence, our performance model merely includes transmission and transcoding costs (t_{trans} and t_{prep}). Two issues remain to be resolved: how to compute t_{prep}?, and how to calculate the expected response time (t_{needed})? The second issue is discussed in Section 4.4.2.

To compute t_{prep} the framework requires that each transformation algorithm *algo* registered[4] provides a function *prepare_costs(algo,obj,param)* returning an estimate for the costs, c_{prep}, incurred by transforming object *obj* from its original quality state to the one currently assigned

[4] As will be detailled in Section 4.6 an application can register with the framework the transformation algorithms to be used for each of the media types supported.

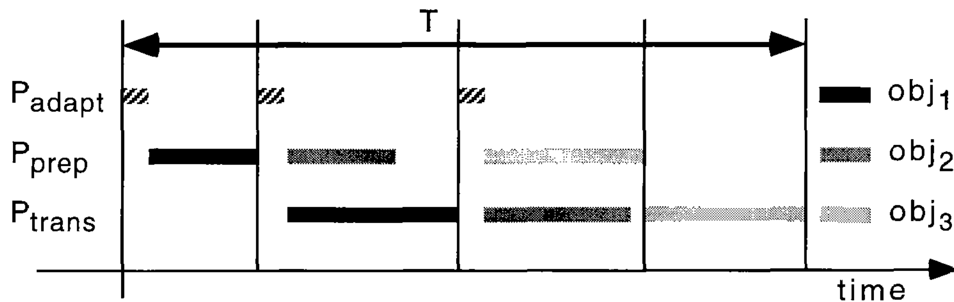


Figure 4.1: Interoperation of phases P_{adapt}, P_{prep} and P_{trans} for a request of three objects.

by the decision making algorithms (Section 4.5). The "target quality" is reflected by the transformation parameter *param*. c_{prep} denotes the costs in terms of resources used, e.g., as given by *system* and *user CPU time* on Unix systems. c_{prep} is used to compute an estimate of the effective t_{prep} needed for a transformation by using an operating system dependent function $prepare_time(c_{prep}, load(t))$. $load(t)$ denotes a prediction for the host's average computational load in the interval $[now, now+t]$. Accurate prediction of host load is beyond the scope of this dissertation. We refer to related work, e.g., by Dinda et al. [41, 42], for further information on this topic. For Unix systems with best-effort scheduling, the time t_{prep} needed to complete a computation-intensive task that consumes c_{prep} CPU time at a system load of $load$ can be approximated by $c_{prep} \cdot load(t)$.

4.4.2 Communication latency hiding

The performance model, which models the response time (t_{needed}) given t_{trans} and t_{prep} depends on how the phases P_{prep} and P_{trans} interoperate. In a simple implementation of the software control loop, the phases of the framework execute sequentially. In reaction to a bandwidth drop, adaptation produces stable results if $t_{prep} + t_{trans}$ for the adapted objects is smaller than t_{trans} for the original objects. However, sequential execution of the phases wastes bandwidth while the host is busy preparing the next object for transmission and wastes CPU resources while transmitting objects over a slow end-to-end path. With a slow connection, the sender is almost constantly congestion-controlled, and there are ample CPU cycles. An improved implementation of the control loop tries to keep P_{trans} constantly sending and uses threaded prepare and transmit phases to hide the latency of the object transformations. *Communication latency hiding* calls for a different response time model: t_{needed} is approximated with $f \cdot t_{prep} + max(t_{trans}, (1-f) \cdot t_{prep})$, where f denotes the fraction of t_{prep} that is not available for latency hiding [179]. For the sake of simplicity and because this is often a realistic situation for applications that are bottlenecked by the network, we assume $f = 0$.

Figure 4.1 schematically depicts how the three phases P_{adapt}, P_{prep} and P_{trans} interoperate. Object $i+1$ is prepared/transcoded while object i is being transmitted. The two phases P_{prep} and P_{trans} synchronize before P_{adapt} is invoked. P_{adapt} operates exclusively. The (small) overheads of the adapt phase are not hidden with communication because P_{adapt} must first decide what data is to be transmitted and in which quality.

There are two additional aspects worth noting in Figure 4.1. First, adapt phases make adaptation decisions based on all objects (and not just a single object) that await transmission by the

Program 4.1: Function $compute_t_{diff}(request, bw(t), load(t), t_{left})$ returning t_{diff}.

$$\begin{aligned}
d_{left} &= \sum_{obj \in request} data(quality(obj)) \\
t_{trans} &= transmit_time(d_{left}, bw(t)) \\
c_{prep} &= \sum_{obj \in request} prepare_costs(algorithm(obj), obj, param(obj)) \\
t_{prep} &= prepare_time(c_{prep}, load(t)) \\
t_{needed} &= overall_time(t_{prep}, t_{trans}) \\
t_{diff} &= t_{needed} - t_{left}
\end{aligned}$$

time P_{adapt} is invoked.

Second, objects are prepared in sequential order, and P_{prep} is only started after both the preceding P_{prep} and P_{trans} phases have finished (e.g., in Figure 4.1, P_{prep} for obj_3 is deferred until P_{prep} for obj_2 and P_{trans} for obj_1 have finished and the adaptation decisions have been revised). The rationale behind this procedure is that adaptation decisions should be deferred as long as possible to minimize the risk of making wrong decisions, that is, decisions that incur unnecessary transformation costs. A second reason for this choice of the *phase schedule* is that it can be implemented easily. We note that such a simplistic phase schedule may be far from optimal because unnecessary idle times may be incurred, during which no data is transmitted (e.g., in Figure 4.1 after P_{trans} of obj_2). Clearly, more sophisticated scheduling schemes could be devised that try to re-arrange the objects with the objective to minimize the communication idle time.

4.4.3 Performance model

Program 4.1 summarizes the performance model that lies at the heart of the control loop's model-based adaptation process (to be described in Section 4.5). The program outlines the steps involved in computing the error variable t_{diff} that drives the adaptation process. The function $compute_t_{diff}()$ takes the request, i.e. the list of objects not yet transmitted, the functions $bw(t)$, $load(t)$, and the time t_{left} remaining for the response delivery as arguments. $bw(t)$ and $load(t)$ model future network and end-system resource availability. $compute_t_{diff}()$ employs two application-specific functions that model the application's resource demands. $data()$ calculates an estimate of the size of an object at a given level of quality (see Section 4.2). $prepare_costs()$ produces estimates for the costs incurred to transcode an object from its original version to the quality specified (by the quality state). Both functions must be supplied by applications that are built upon the framework (see Chapter 5).

Although the intrinsics of different inter-operation schemes and the various resource models are outside the scope of the framework, the discussion in the previous sections emphasizes the need for suitable abstractions. To allow for future refinements and extensions we encapsulate the performance model for the response time by a set of functions, such as *transmit_time(), prepare_costs(), prepare_time()* and *overall_time()* that can be used to compute t_{needed}.

Program 4.2: Pseudo code for function $adapt(request, bw(t), load(t), t_{left})$.

```
t_diff = compute_t_diff(request, bw(t), load(t), t_left)
try
    if( t_diff > ε ) then            // prevent network overload
        t_diff = reduce(request, bw(t), load(t), t_left);
    elsif( t_diff < -ε ) then        // prevent under-utilization
        t_diff = expand(request, bw(t), load(t), t_left);
    if( |t_diff| > ε ) then
        throw new NoAdaptationPossible();
catch( NoAdaptationPossible exception )
    handle (exception);              // application specific handler
end
```

4.5 Quality-aware decision making

The performance model captured by $compute_t_{diff}()$ is used by the function $adapt()$, which is sketched in Program 4.2, and is invoked repeatedly by P_{adapt} after obtaining new bandwidth feedback $bw(t)$. If t_{diff} strays out of the tolerance window, that is, if t_{diff} exceeds an application-specific threshold ε, adaptation is required and the remaining objects in the *request* are subject to the adaptation process described in the next sections. ε reflects the user's (or application developer's) tolerance on how much the response time can deviate from the time limit specified. ε helps to limit oscillations and hence potential instabilities in the adaptation process.

To accomplish the adaptation, the sender must address the three issues described in Section 4.3. Given the list of objects that must be transmitted, there are several possible approaches to identify the victims, distribute the quality reduction, and select the transformation algorithms. We discuss two such approaches in the following sections.

Independent of the concrete decision-making procedure is how the application deals with situations where adaptation is not possible. E.g., if the adaptation potential of the objects in the request is too small, then an application-specific exception handler is invoked that decides how to deal with the situation (see Program 4.2). E.g., the handler could invoke a user-dialog either to inform the user that the time limit cannot be respected or to renegotiate the time limit (Section 4.9).

4.5.1 Generic solution

To avoid congestion and network under-utilization, the adaptation process should aim at finding a combination of objects to adapt, and transformations to apply, such that $|t_{diff}|$ is minimized and the overall quality metric is maximized. Unfortunately, an exhaustive search for the global minimum of $|t_{diff}|$ in the whole solution space is not attractive, as we illustrate in the next paragraphs.

Given a request consisting of N objects, given that an average of M transformation algorithms are applicable to adapt the quality of these objects, and given that each of these algorithms take an average of m different parameter values, there are $n \approx \sum_{i=0}^{M} \binom{M}{i} m^i$ possible

Program 4.3: Pseudo code for function $reduce(request, bw(t), load(t), t_{left})$.

```
victim = choose_victim(request);
do
    state = save_state(victim);
    do
        reduction possible = reduce_victim(victim);
        t_diff = compute_t_diff(request, bw(t), load(t), t_left);
    while( ( no reduction of t_diff achieved ) && ( quality reduction possible ) );
    if( no reduction of t_diff achieved ) then
        reset_state(victim, state);
        fi
    victim = choose_victim(request);
while( ( ∃victim ) && ( t_diff > 0 ) );
return t_diff;
```

transformations applicable to each of the N objects. If we assume that all the possible combinations fulfill the boundary conditions on object quality, there are approximately n^N possibilities to adapt the request to the currently available bandwidth. In each iteration of P_{adapt}, the sender must compute t_{diff} for each of the n^N points in the solution space. For all the combinations with $|t_{diff}| < \varepsilon$ the sender must compute the overall quality as the weighted sum of relevance scores and utilities of the individual objects (Section 3.3) and choose the one with maximal overall quality. This decision making algorithm is simple and able to produce well-founded quality-aware decisions as it covers all of the issues listed in Section 4.3 in an integral approach. However, the run-time complexity may be fairly (or even intractably) high.

As long as there is no additional information about the functions used to compute t_{diff} (e.g., gradients that may direct the search), or as long as the quality boundaries are not very restrictive, the size of the solution space cannot be reduced, and hence the complexity is too high to make this approach feasible in the general case. Therefore, we cannot base the framework's adaptation mechanisms on a generic method that performs an exhaustive search, since we expect the methods of the framework to provide a solution for all possible extensions. However, we can provide the application with several *strategies* [65] for the adaptation process (one being exhaustive search for example) and leave it to the application developer to decide on the most appropriate strategy to use in the context of the application.

4.5.2 An approximative solution

If N or n are large, the sender must employ some approximations or introduce simplifications in the adaptation process to reduce the complexity of the adaptation process. Otherwise, the search is so expensive that the resource consumption of P_{adapt} can no longer be neglected. High decision making overheads render adaptation useless.

The idea that forms the basis of our approximative solution is that different transformation algorithms are likely to have different impacts on the quality (or utility) of an object (but may exhibit similar adaptation potentials). If the application (or the user) provides the framework

Program 4.4: Pseudo code for function *choose_victim(request)*.

```
best_metric = −∞;
foreach ( obj ∈ request ) do
    state = save_state(obj);
    if( reduce_victim(obj) ) then
        Δsize = data(quality(state)) − data(quality(obj));
        Δutility = utility(quality(state)) − utility(quality(obj));
        metric = compute_metric(Δsize, Δutility);
        if( metric > best_metric ) then
            best_metric = metric; victim = obj;
        end
    end;
    reset_state(obj, state);
od; return victim;
```

with a prioritized list of transformation algorithms (high priority implies low impact on utility), then the adaptation process can approximate the search for a minimal $|t_{diff}|$ by iteratively trying to apply the possible transformation algorithms with their respective parameters with the objective to find a local minimum that is within the tolerance. If one algorithm does not achieve the desired result, the next algorithm is chosen. To resolve the issues mentioned in Section 4.3 the adaptation phase proceeds along the steps outlined in Program 4.3, which exemplarily sketches the *reduce*() function invoked in Program 4.2; the *expand*() function works similarily.

The function *reduce*() in Program 4.3 repeatedly chooses a *victim* and tries to reduce the quality of the chosen *victim* until t_{needed} is in line with t_{left}, that is, until t_{diff} reaches zero. The process of reducing the quality of a victim iteratively tries to find a state that achieves a reduction of t_{diff}. Quality reductions of an object are only commited, that is, the quality state of the object is only changed, if it results in a reduction of t_{diff}. This procedure has been chosen to prune the solution space and to avoid situations where the transcoding costs (t_{prep}) exceed the transmission costs (t_{trans}) for the particular object. The function *reduce*() relies on two important auxiliary functions. First, the function *choose_victim(request)* is responsible to find a *victim* for quality reductions that satisfies the requirement of having a low impact on overall utility of the response. Second, the function *reduce_victim(object)* is responsible for quality reduction of an individual object. Quality reductions on an object are achieved "stepwise", that is, by repeatedly invoking the function *reduce_victim(object)*.

Program 4.4 illustrates the function *choose_victim*() which identifies among the objects that are requested but not yet delivered an ideal candidate for quality reductions. Whether the quality of an object should be reduced depends on the utility of the object to the user and on the size reduction achievable. To get an estimate of the impact on the metrics response time and utility that a quality reduction of a particular object may have, we fictitiously carry out the "next" reduction step for this object and compute the difference in utility ($\Delta_{utility}$) and size (Δ_{size}). To accomplish a single reduction step the function *reduce_victim*() is employed (see below). The metric that steers the choice of a victim is caclulated by the function $compute_metric(\Delta_{size}, \Delta_{utility})$. This function allows for several interpretations (or victim selection strategies [65]) that can be chosen

Program 4.5: Pseudo code for function *reduce_victim(obj)*.

```
foreach( algo ∈ applicable_algorithms(type(obj)) )while( no reduction achieved )do
    if( is_reducible(obj,algo) ) then
        param_iter = transform_param(algo,original(obj),current(obj));
        advance(param_iter);
        target_quality = target_quality(algo,original(obj),param_iter);
        if( minimum_quality ≤ target_quality ) then
            set_state(obj,target_quality);
            // how to transcode? costs? size?
            find_version_match(obj,target_quality,version_cache);
            if( no matching version found in cache ) then
                find "best" base version and algorithm sequence (algo_chain)
                that transcodes the base version into targeted quality state;
                "best" = version/algo_chain that incurs smallest costs
            end // otherwise, no costs incurred
        end
    end
end; return reduction achieved;
```

by the application developer. E.g., the object with the lowest impact on utility can be chosen ($min_{obj \in request}(\Delta_{utility})$), or an object with low impact on utility but high potential for size reduction can be selected (e.g., $min_{obj \in request}(\Delta_{utility}/\Delta_{size})$), etc.

Program 4.5 sketches how an individual object is reduced in quality. As mentioned above, the problem of ambiguity in selecting an appropriate transformation algorithm for a particular quality reduction is resolved by the priorization of the transformation algorithms applicable to the object at hand. The algorithms are applied in the order specified by the application (developer). If the object is reducible with respect to the selected transformation algorithm (*is_reducible()*), the parameter level that transcodes the object from its original version to its currently assigned quality state is looked up. The next reduction step is produced by advancing in the list of applicable parameters for the transformation at hand. If the resulting target quality does not violate the restrictions on minimum quality, the new object quality can be assigned.

Then, the algorithm sequence that transforms the object from its current quality to the target quality state must be determined. We use the term algorithm sequence (*algo_chain*) to indicate that several transformations may be necessary to achieve the desired quality reduction. Transformations can be expensive in terms of CPU usage. Therefore, it may be beneficial to cache (intermediate) results of such transformation steps, that is, to store different versions of an object, in a *version cache*. The algorithm sequence that incurs the smallest transformation costs should be chosen. For this purpose, we first try to find in the version cache a version of the object to be delivered that matches the newly assigned quality state. If such a version is found, no transformations are required and $c_{prep} = 0$. If no matching version is detected, the cache is searched for the version that is best suited to serve as base version for the transcoding. The prepare costs decide about the usefulness of base versions. Finally, the algorithm sequence that transforms the object from its base version to the target quality state is determined and recorded.

4.6 Framework structure

The previous sections in this chapter have detailed the decision making process/algorithm and have therefore described the dynamic behavior of the P_{adapt} phase that lies at the heart of our application framework. This section first describes the static structure of the framework, that is, the data structures employed. Second, it briefly describes how the data structures collaborate to carry out the tasks of request processing, P_{adapt}, P_{prep} and P_{trans}.

4.6.1 Data structures

The structure of the core framework is illustrated with the UML class diagram shown in Figure 4.2. The class diagram is drawn from a *specification perspective* as defined by Fowler et al. [56]. The focus lies on interfaces; implementation details, such as query and modifier methods to read and write attributes, are omitted.

We introduce a number of notational issues: the name of abstract classes and abstract methods is shown in italics. Those abstract classes that must be extended by applications derived from the framework (see Chapter 5) are shaded. Attributes marked with '/' are so-called *derived attributes*, i.e., attributes that can be calculated from other associations and attributes on the class diagram. Arrows on association lines indicate navigability [56]. Association roles are labelled at the target class. If there is no label, the role is named after the target class. The following paragraphs describe the classes, their attributes and associations. The next section sketches how these classes are put to use.

The class *RequestEntry* represents requested objects to be delivered by the server. Each request entry has a *type*, e.g., text, image, video, etc. The different versions of an object that are stored in the server's repository or in the version cache are summarized by the *version list*. A *version* is characterized by its storage *location* and its *quality*. Versions can be queried for their *size*. The *quality* captures *type*-specific attributes and provides a simple quality metric through the method *utility()* (Section 3.3). The *utility* of an entry is a function of its *relevance* score and the utility of its currently assigned quality. The attributes *minimum* and *maximum quality (minQ, maxQ)* reflect the restrictions on object quality imposed by the client. An entry's adaptation *state* captures the currently assigned quality (*current*), a base version (*original*) and a so-called *algo_chain*. The *algo_chain* comprises a sequence of algorithm and parameter pairs and describes how—starting from the base version—the object must be transcoded to achieve the targeted quality (*current*). The *algo_chain* is also used to produce estimates of the prepare costs and the size of the *final* version of the object.

The abstract class *Type* is central to the framework as it provides abstractions for many of the specifics to be filled in by applications derived from the framework. E.g., it knows which quality-subclass must be instantiated and helps to find version matches and to determine which combination of algorithms can transcode an object from its original quality to the targeted quality (see Program 4.5). Furthermore, the *type* determines which transformation algorithms are applicable to the objects of this type[5].

Algorithms transform objects from an *input* type (or encoding) to an *output* type. An algorithm comprises two resource models, a *cost model* to estimate the CPU-costs of applying the

[5]For the sake of simplicity, we do not show how the framework deals with different *encodings* of a media type.

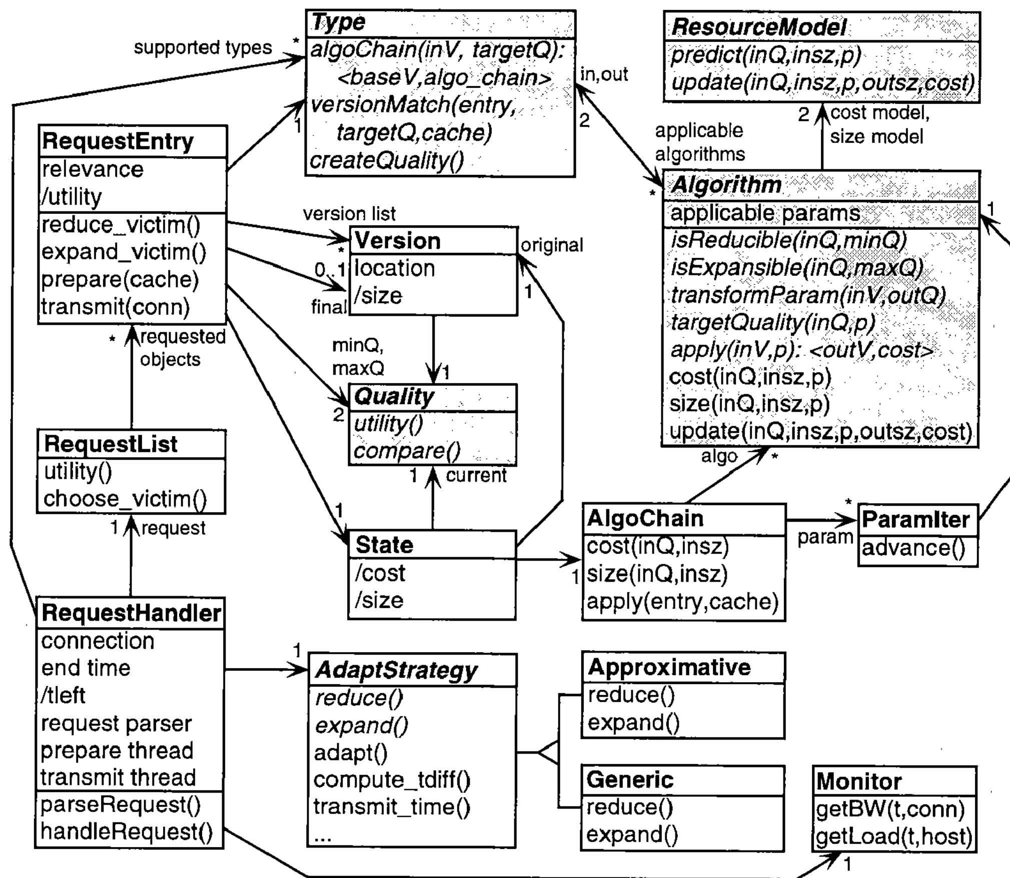


Figure 4.2: Class diagram of framework core (in UML notation [56]).

algorithm to an object, and a *size model* that estimates the size of a transformed object. Note, that there is a discrepancy between the notation used in previous sections (e.g., Section 4.2) and the class diagram. The previous sections postulated a function $data : Quality \rightarrow \mathbb{N}$ for the purpose of estimating the size of an object of a given quality. Note, however, that changes in quality are either achieved by choosing an appropriate version of the object (e.g., from the cache) or by transforming the object. Since size information for versions is readily available, associating the size model directly with the algorithm achieving the quality changes seems a natural choice.

The class *RequestHandler* provides the "glue" to the classes mentioned above. It encapsulates the control loop with its threaded *prepare* and *transmit* phases, drives the *adaptation strategy* based on information about resource availability obtained by a *monitor*, and has a handle on the server's *repository* and *version cache* (not shown in the diagram). Furthermore, the application registers all the *supported (media) types* with a prototype request handler (Chapter 5). The application-specific *request parser* retrieves request messages from the client *connection* and builds the *request list* which contains a *request entry* for each of the objects requested.

4.6.2 Interaction of data structures

This section briefly discusses how the data structures are put to use (i.e. interact) for the request processing and the three phases of the control loop.

A new client connection arriving at the server is handed off to a request handler, which retrieves the request message and parses it as mentioned above; the method *parseRequest()* delegates the parsing to an application-specific request parser. The request list is built, and the user-specified time limit is translated to a deadline (*end time*). The attribute t_{left} can be derived from the *end time*. The method *handleRequest()* spawns the prepare and transmit threads, which operate (and synchronize with the main adapt thread) as depicted in Figure 4.1. The control loop is entered and each time P_{adapt} is activated, the performance monitor is polled for the latest information about network and end-system resource availability and the adaptation strategy's *adapt()* method is invoked (Program 4.2).

P_{adapt} then operates as described in Section 4.5 and as outlined in Programs 4.1–4.5. The functional notation used in the pseudo code can be translated to the object-oriented design, that is, to the class diagram as follows. $foo(obj)$ or $bar(class)$ indicate that the method *foo* is invoked on object obj or that the method *bar* is invoked from class $class$. Program 4.1 is implemented by the $compute_t_{diff}()$ method of *AdaptationStrategy*. The *reduce()* method shown in Program 4.3 is provided by the class *Approximative*. The function *choose_victim()* (Program 4.4) is implemented in the class *RequestList*. Finally, the class *RequestEntry* provides the *reduce()* method shown in Program 4.5. This method heavily relies on the abstractions provided by the classes *Type* and *Algorithm*. Subclasses of *Type* must provide the methods *versionMatch()* (named *find_version_match()* in Program 4.5) and *algoChain()*; *Algorithm* must fill in *isReducible()*, *create_param_iter()* and *targetQuality()* to make the generic *reduce()* function work. The decision making algorithms and the performance model for the response time have been thoroughly discussed in the previous sections. In addition, the only aspect worth noting is how estimates for the transformation costs are calculated ($prepare_costs()$ in Program 4.1). The method *cost()* of *AlgoChain* iterates through all the algorithm-parameter pairs and sums up the costs incurred by all transformations. In each step it estimates the CPU costs for applying

the algorithm on the object to be transformed (method *cost()* of *Algorithm*) and computes the resulting quality of the object (method *targetQuality()*). This step is needed to compute the costs of the next algorithm in the *AlgoChain*. The actual cost estimation is delegated to the *cost model* supplied with the algorithm (method *predict()*). Size estimation works accordingly. The resource models are discussed in more detail in Section 5.1.3.

P_{prep} and P_{trans} invoke the methods *prepare()* and *transmit()* on the request entry to be transformed or delivered. The transformation works as follows. The method *apply()* in the class *AlgoChain* iterates through the algorithm-parameter pairs listed. In each step the method *algorithm.apply()* is invoked, which produces a new quality-reduced version of the original object. New versions are added to the cache. After each transformation step the CPU costs incurred and the size of the resulting version are fed to the algorithm's resource models to allow them to *update* their models. After the last transformation step the request entry is updated with the location and size of the *final* version. The final version is used by the *transmit()* method to deliver the object.

4.7 Start-up behavior

So far, this chapter has described a framework for network-aware applications and shown how such applications can dynamically adjust their demand of network resources to match the supply of these resources. To avoid burdening the description of the core concepts with additional complexity, discussion of adaptation inherent problems, such as start-up behavior, agility, etc. has been deferred to the following sections. This section defines the "start-up problem" and describes solutions that help alleviate it. Section 4.8 discusses concerns about adaptation granularity, and Section 4.9 addresses miscellaneous issues barely covered in the previous sections.

4.7.1 Problem

Feedback control systems are typically faced with the problem of finding the optimal operating point as soon as possible after start-up. Thereby, special care must be applied because both overshooting and excessively conservative, i.e. slow, start-ups should be avoided mainly for reasons of performance. This observation also applies to network-aware applications. Thus, the question is: How can an adaptive application find the appropriate operating point quickly? For network-aware applications, we distinguish the two cases of performance-based and model-based adaptation (see Section 3.4.1).

First, the reactive nature of performance-based adaptation, which does not make use of explicit information about network status, requires that the application must use "probing" to learn about resource availability. The only reasonable way of doing so is to start out conservatively and to increase the sending rate slowly, and to increase it only after learning that the network can sustain the current sending rate. Congestion-aware transport protocols, such as TCP, are faced with the same problem at start-up. In fact, it is by means of such a conservative probing mechanism (called slow-start) that a TCP sender discovers the available bandwidth [79, 176].

Second, model-based adaptation could—at least in principle—zoom in quickly on the appropriate control parameters and would not require such (conservative) "probing" mechanisms to learn about available resources (Section 3.4.1). However, the adaptation decisions depend

on the knowledge of the available bandwidth—information which is typically not available at the start of a connection. Thus, the start-up problem for such applications manifests itself as follows.

On one hand, the application should start to send (useful) data right away for two purposes: first, to get information about network status as quickly as possible, and second, to utilize the communication channel and not to waste resources. On the other hand, the application can decide only after getting information about network service quality how much data (in which quality) to send. Thus, sending application data right away bears the risk of missing the time limit. To resolve this conflict, a tradeoff between maximizing the utilization of the communication channel and minimizing the risk of violating the time limit must be found. The following section sketches how such a tradeoff can be achieved.

4.7.2 Mechanisms

This section discusses mechanisms that help alleviate the start-up problem of network-aware applications. We distinguish two groups of approaches: first, solutions that allow to minimize the start-up inherent risks (without sacrificing high resource utilization), and second, mechanisms that attempt to speed up the resource discovery process. Section 4.7.3 shows how these approaches are integrated in our framework.

Risk minimization

The first group of mechanisms attempts to reduce the start-up inherent risks. What are these risks? First and foremost, there is the risk of missing the user-specified time limit. This situation can occur if the application starts sending data without knowing in which quality to send it. If the first object to be delivered is large and bandwidth turns out to be unexpectedly low, the application may risk to miss the deadline. Or it may at least have to deal with the fact that the delivery of the first object consumes a disproportionate amount of the time frame allotted. This situation leads to the second concern: by starting object delivery in an uninformed way, the application risks to waste adaptation potential which may then be missing at the end of the transfer.

Berger [13] devised and implemented two techniques in the context of our framework that reduce these risks: *reordering* of objects within the request, and increasing *agility* by means of hierarchical encoding and progressive delivery. Issues related to agility are detailed in Section 4.8. Reordering objects within a request allows to reduce the risk of being stuck with a large object or with an object that wastes precious adaptation potential during start-up[6]. Reordering requires that the client must be able to deal with out-of-sequence delivery of the objects requested.

How should the request list be reordered? Different criteria can be applied when picking the object to be delivered next from the list of requested objects. First, objects that do not allow for a quality-size tradeoff or that have a low adaptation potential are good candidates for transmission

[6]In addition to alleviating the start-up problem, reordering can also be useful in cases where bandwidth information is available (e.g., from a bandwidth cache). In such situations reordering may help to avoid the initial idle period in the transmit phases (see Figure 4.1), e.g., by selecting and transmitting an object that does not need to be transformed.

during start-up, because they must be transmitted (almost) unchanged anyway. Second, small objects are an attractive choice, because they increase the chance that the control loop can take corrective action, if need be, and because small objects usually exhibit only small adaptation potential. Likewise, objects should be picked that allow for progressive delivery. And finally, if rate information is available and reordering is only employed to fill an initial gap in the transmit phase (as shown in Figure 4.1), an object that incurs no (or only marginal) prepare costs should be selected for prioritized delivery.

Resource discovery

There are two complementary approaches to obtain trustworthy bandwidth estimates quickly.

Bandwidth caching. Although there are many paths in the Internet that exhibit significant fluctations of the available bandwidth, Balakrishnan et al. [10] and Paxson [141] observed that there is also a significant fraction of Internet paths whose performance is fairly stable over ranges up to tens of minutes. This observation suggests that caching the performance of completed client sessions (connections) could be beneficial, as it would allow to predict the available bandwidth for new connections from the same client or from nearby clients by extrapolating from past measurements. The SPAND system [163] implements this idea. The main objective of the SPAND system is to aid clients in selecting those servers from a set of mirror servers which promise best performance. (For further references on server selection, see [126]). A similar approach can also be applied at the server side to help a network-aware application (derived from our framework) to quickly get an estimate of available bandwidth. The toolkit for resource prediction developed by Dinda et al. [43] can be used to choose the appropriate prediction model. Resource predictions are provided with confidence intervals that indicate their trustworthiness of the predictions.

Bandwidth probing. If a request arrives at the server from a "first-time" client, or from a client for which only out-dated bandwidth information is cached, the network-aware sender may want to probe for the bandwidth available before making adaptation decisions and before sending application data. In this context, probing means that the application transmits and times a few (usually MTU-sized) packets and tries to infer estimates of available bandwidth from the timings. A number of related techniques to probe for bandwidth availability have been proposed in the literature, most of them in the context of server selection, e.g., [29, 28, 104, 92, 141]. The problem with bandwidth probing is that it incurs considerable overheads (it delays the application in sending useful data and it stresses the network). However, there are some situations where probing provides the best solution to minimize the risk of missing a deadline (see below).

4.7.3 Discussion

The mechanisms presented in the previous section are complementary when it comes to defusing the start-up problem of network-aware applications and hence can all be integrated into the framework. The flow chart depicted in Figure 4.3 illustrates how these techniques are put to

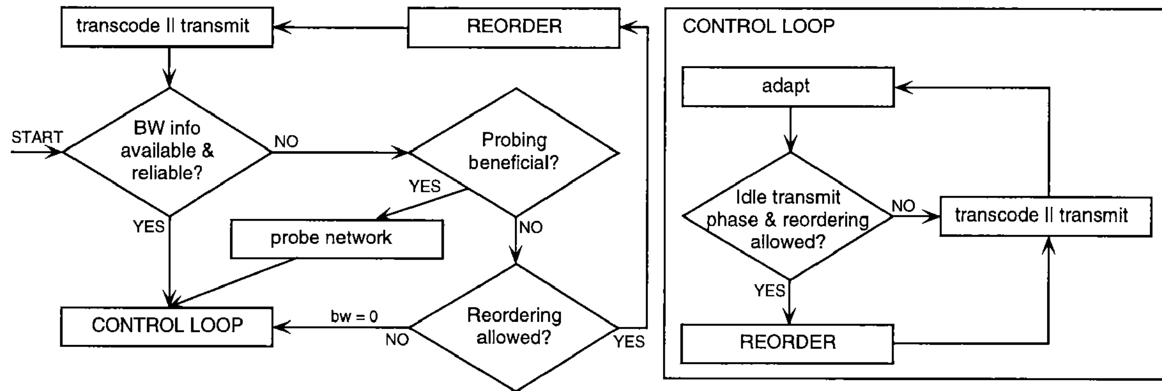


Figure 4.3: Integration of start-up mechanisms (flow chart).

use in our framework. The *reorder* algorithm sequentially tries to apply the criteria listed in the previous section to pick the object to be transmitted next. If an object matches one of the criteria then the prepare and transmit phases are activated to deliver the object. Two issues remain to be resolved (to understand the flow chart). How can we decide whether a bandwidth estimate is trustworthy? When is bandwidth probing beneficial?

To answer the first question, let T be the time limit, bw be an estimate for the bandwidth available, and let bw_{min} denote an estimated lower bound for the bandwidth available. (For example, bw and bw_{min} may be obtained as predictions from past measurements stored in a bandwidth cache.) Recall that the adaptation algorithms (Section 4.5) base their decisions only on the bandwidth estimate bw. The estimate bw is termed trustworthy if the following condition holds: the application is able to meet the time limit regardless of whether the actual bandwidth is indeed bw or whether it is only bw_{min}. On the other hand, the bandwidth estimate bw is *not* trustworthy if the following condition holds: the application is able to meet the time limit if the actual bandwidth is indeed bw, but the application misses the time limit if the actual bandwidth is bw_{min} (even if all objects are reduced to minimal quality). This condition can be fairly easily evaluated using the decision-making mechanisms employed by the framework (see Section 4.5).

To address the second question, let s_{small}, s_{min} and s_{probe} denote the size of the smallest object requested, the size of the smallest object when reduced to minimal quality, and the amount of data that must be injected into the network (by a probing mechanism) to form reliable bandwidth estimates[7], respectively. Probing is clearly not advisable if $s_{small} \leq s_{probe} + s_{min}$. (In this case, the probing overhead exceeds any potential gain that could be derived from knowing the available bandwidth.) Furthermore, the overheads incurred by probing are justified only if the smallest object is so large that with a conservative estimate of bandwidth the time limit is likely to be missed, that is, if $s_{small} > T \cdot bw_{min}$.

How can the effectiveness of mechanisms dealing with the start-up problem be assessed? Three metrics are important. The first metric is the time it takes until reliable information about available bandwidth become available *and* the application is able to adapt. The latter point is important because there is no use of knowing bandwidth and not being able to take corrective action. Second, the utilization of the available bandwidth (with application data) during the start-up phase should be high. Third, as little adaptation potential should be wasted as possible

[7]Here, we assume that probing is performed with "useless", i.e., non-application relevant data. If the application is highly agile (see Section 4.8) probing can be performed with application data, in which case $s_{probe} = 0$.

until bandwidth information is available (to not compromise the flexibility of the adaptation process). Berger [13] evaluated the start-up techniques discussed and found that (substantial) improvements for all three metrics can be observed compared to a system that does not address the start-up problem.

Related work on network-aware applications (see Section 2.3) has hardly addressed the problems related to start-up. Noble's evaluation of Odyssey [130] only considered adaptation after the system reached steady-state. The proxy-based approaches pursued by Fox et al. [59] or Han [70] merely use a SPAND-like bandwidth cache. Video distribution by means of RLM [113] employs performance-based adaptation and hence goes through similar steps as TCP during its slow-start.

4.8 Agility

The model of adaptation employed by the control loop in our framework may be problematic, because adaptation decisions can only be reconsidered after P_{prep} and P_{trans} have finished transcoding and delivering the current object. The latency of these prepare and transmit activities may pose a problem because P_{adapt} must rely on either good bandwidth estimates or on the expectation that network service does not degrade more during the next phases P_{prep} and P_{trans} than there is data reduction potential inherent to the remaining objects in the request list. Due to the nature of best-effort network service, these assumptions may not always be fulfilled. As a consequence, if the application is not *agile* enough to react to drastic changes in the network environment, such situations may occur and may result in the breakdown of the service model.

Application *agility* can be defined as the speed with which the application can react to changes in resource availability. Agility depends on two independent aspects: the speed with which the adaptive application can *detect* changes in resource availability, and the speed with which it can *react* to such changes. This distinction is often ignored because many adaptive systems are customized for a single application, but it is important for a framework for network-aware applications. Following the argumentation of Chapter 3, we can separate the two concerns of agility. The speed with which changes are detected depends on the techniques employed for resource discovery and is therefore discussed in Chapters 7–9. The detection speed imposes an upper bound on how fast the application can react to changes. However, it is often the case that agility is limited by the application because the granularity at which adaptation decisions can be made is coarse. Here, granularity indicates how closely the adaptation points (P_{adapt}) are spaced timewise.

4.8.1 Progressive delivery

Agility can be improved by increasing the number of adaptation points. With the current scheme there is one adaptation point per object. Consequently, agility can only be improved by using multiple adaptation points per object. The current scheme assumes that each object is an entity for encoding. Thus, with this scheme, there is no use in setting adaptation points earlier than at the end of a completed object transmission because revising adaptation decisions earlier wastes network and CPU resources. As a consequence, multiple adaptation points per object can only be had by using sophisticated coding schemes. Hierarchical coding schemes, such as described

Program 4.6: Progressive object delivery (function $drainScanQueue(scan_list, data, T_{limit})$).

```
t_to_send = T_limit - now;
while( ¬empty(scan_list) && ( is_active(P_prep) || t_to_send > 0 ) ) do
      scan = pop(scan_list);
      t_to_send = send_scan(connection, scan, T_limit);
      data = data - size(scan);
      bw = getBW(monitor, connection, T_limit);
      t_to_send = min(t_to_send, data/bw);
end
if( ¬empty(scan_list) ) then
      append the remaining scans to the request handler's scan list
end
```

in [77, 5] for image data or in [115] for video data, break the data into serveral layers (or scans). The first layer encodes the most significant information, further scans add "deltas" that refine the quality of the object. It is important to note that any number of scans delivered in sequence are self-contained and provide an independent—albeit quality-reduced—version of the original object. This property allows to deliver hierarchically encoded objects *progressively*, that is, one scan after another and to terminate delivery of an object after each scan, if need be. Thus, the number of adaptation points for each object is increased by the number of layers in the coding scheme. Hierarchical encoding and progressive delivery has also been applied in other adaptive systems, e.g., [113, 152, 66, 98].

Progressive object delivery has been implemented in our framework as follows [13]. Objects that are hierarchically encoded have an attribute *scan_list* which represents the list of scans (or layers) of the encoded object. A scan provides a method to uniquely identify itself within an object and return the scan's data. The request handler also contains a *scan_list*, that is, a list of all undelivered scans. The additional adaptation points gained by progressive, i.e. scanwise, delivery of hierarchically encoded objects are not used to directly invoke P_{adapt} after each scan (because of the decision making overhead), but are merely used to control the P_{trans} phase and to terminate delivery of an object, if bandwidth drops drastically. The phase P_{trans} computes a limit T_{limit} for the delivery of the current object based on the object's size ($data$) and a bandwidth estimate. It then invokes the function $drainScanQueue()$ shown in Program 4.6 with the object's scan list. $drainScanQueue()$ serves three purposes: it allows reaction (i) if bandwidth drops, or (ii) in case of a sudden abundance, and (iii) it tries to avoid transmission idle periods incurred by P_{prep}/P_{trans} mismatches. The function transmits scans as long as there is time left for the object delivery or the concurrently operating prepare phase is still active. The first condition protects the application from wasting a disproportionate amount of network resources on the currently delivered object in case of a bandwidth drop. The second condition is to prevent transmission lulls that can occur if $t_{prep} > t_{trans}$. Although t_{to_send} may be less than 0, the prepare phase may still be active and thus, P_{adapt} cannot be invoked. Moreover, after each scan the monitor is queried for an update of the bandwidth information and t_{to_send} is recalculated to prevent spending too much time on transmission of the current object in case bandwidth increases. If the loop condition is no longer satisfied then the remaining scans are appended to the request

handler's list of unsent scans. After returning from *drainScanQueue*() on the object's scan list, the function *drainScanQueue*() is invoked for the list of unsent scans. The latter call is also made after the transmission of a non-hierarchically encoded object. Draining the request handler's list of unsent scans, in case P_{prep} takes longer than anticipated, helps to maintain a high utilization of the available bandwidth and hence to maximize the quality of the content delivered to the client.

4.9 Discussion

This section briefly discusses adaptation-related issues that have received little attention in the description of our framework so far.

Boundary conditions/Breakdown of service model. In addition to start-up and agility related problems, ill-specified boundary conditions pose another threat of failure that requires application-specific reaction. No application should set a (short) time limit T and then require a high minimal quality such that even sending at minimal quality exceeds the time limit. However, the appropriate settings of the boundary conditions cannot always be anticipated. Therefore, an application must be able to deal with such situations that could lead to the breakdown of the service model. Possible reactions include delivery of objects at minimal quality, a user-application dialogue to renegotiate the boundary parameters, or termination of transfers altogether. This last option is attractive if it allows an overloaded server to catch up. The application-provided exception handler in Program 4.2 deals with such situations.

Client processing speed. The service model implies that the response time should be respected on an end-to-end basis. So far, we have neglected that some clients may be resource poor and that presentation conversion of the data delivered may incur significant costs and hence latency. Thus, the question is how can we incorporate client processing speed in our sender-based adaptation framework? There are two issues: first, presentation conversion must be off-loaded to the network-aware sender by appropriately specifying the boundary conditions on object quality. Thereby, the goal must be to off-load as much of the presentation conversion as possible so as not to have the client be the bottleneck in the delivery pipeline comprising the three stages server-side transcoding, network transmission and client-side presentation conversion. Second, if the client is relieved to such an extent that it is no longer a bottleneck, then end-to-end response time can be respected by the framework if the (worst-case) client processing time for a single object (the last stage in the delivery pipeline) is subtracted from the user-specified time limit before the request is sent to the server.

Prediction accuracy. The model-based adaptation process described in this chapter relies heavily on predictions of resource availability. Predictions are rarely fully accurate and the estimates are often accompanied with a confidence interval (as noted in Section 4.2). The repetitive nature of the control loop provides some robustness against inaccurate estimates (see Section 4.2), however, the robustness of the decision making algorithms presented so far could be improved by making use of confidence intervals for resource predictions. We briefly discuss

how confidence intervals, e.g., on the bandwidth estimates $bw(t)$, can be used to improve the robustness of our adaptive system. Let $[bw_{min}(t), bw_{max}(t)]$ denote the confidence interval for $bw(t)$ (at an application-defined confidence level). P_{adapt} uses the function $compute_t_{diff}()$ (Program 4.1) to initiate (Program 4.2) and drive (Program 4.3) the decision making process. Using $compute_t_{diff}()$ the confidence interval $[bw_{min}(t), bw_{max}(t)]$ can be translated into an upper and a lower bound on t_{diff}. There are three alternatives to change the way adaptation decisions are made based on these bounds. First, an optimistic adaptation strategy could always use the lower bound on t_{diff}. Conversely, a conservative strategy would rather use the upper bound on t_{diff}. A third alternative could try to bring the interval $T = [min(t_{diff}), max(t_{diff})]$ in line with the tolerance interval $E = [-\varepsilon, \varepsilon]$ on t_{diff}, such that either $T \subseteq E$ or $E \subset T$ holds, depending on which of the two intervals is wider. Assuming $|T| < |E|$, adaptation is initiated when $T \nsubseteq E$ and terminates only when $T \subseteq E$. This third alternative is a compromise between the optimistic and the conservative approach, and it improves robustness (compared to the decision-making algorithm presented in this chapter) by taking the prediction error into account.

Communication idle time. Gaps in the sequence of object transmissions should not only be avoided because of the transmission opportunities lost at the application level, but also because many congestion control mechanisms exhibit a *use-it-or-lose-it* property [48]. That is, communication idle time results in loss of the fair share of the bottleneck bandwidth previously held by the connection and consequently results in repeated start-up behavior. Such gaps can occur if P_{prep} lasts longer than the concurrently executed P_{trans} phase. Reordering of the objects in the request with the goal to minimize the communication idle time and the progressive delivery of objects outlined in the previous sections can be used to minimize the negative impact of transmission lulls.

4.10 Summary

This chapter describes the key concepts that form the basis of our framework for network-aware applications. Central to the model-based adaptation employed in the framework is a performance model for the response time that incorporates both network *and* end-system resource availability. Based on the response time model, a heuristic adaptation algorithm has been developed that is capable of producing quality-aware adaptation decisions efficiently. Emphasis is put on the design of an adaptation process that is both flexible and reusable by network-aware applications derived from the framework. Reusability is achieved by factoring out three aspects of application-specific functionality: first, the object types handled by the application, second, the algorithms applicable to transcode objects of a particular type, and third, the resource models for the costs incurred and the size reductions achieved by these algorithms. The claims of reusability are reviewed in Chapter 5. The claims of efficiency and quality-awareness are treated in Chapter 6. In addition to describing the core adaptation process, the chapter discusses a number of adaptation inherent problems, such as start-up behavior and application agility, and sketches the solutions provided by the framework.

Chapter 5

Framework instantiation

This chapter pursues two goals. First, it illustrates how the framework described in the previous chapters can be instantiated to construct a network-aware application. The sample application used for this purpose is Chariot, an integrated image search and retrieval system [193]. Second, the framework design is reviewed under the aspect of potential (code) reuse (i) for applications from the same application domain, and (ii) for applications from (slightly) different domains.

5.1 Sample application: Chariot

We illustrate the general principles developed in the previous chapters with examples from a specific project, the Chariot (Swiss (**CH**) **A**nalysis and **R**etrieval of **I**mage **O**bjec**T**s) project. This section briefly introduces the architecture of the Chariot system before proceeding with the description of how the framework is instantiated. In addition, the section describes the Chariot-specific resource models.

5.1.1 System architecture

The objective of the Chariot system is to allow networked clients to search a remote image database. The Chariot system uses query-by-example [51, 40] to let a user formulate a reference for images similar to a given query image. The core of the system (as depicted in Figure 5.1) consists of a client (to handle user access to the image library), a search engine to identify matching images, and one or more *network-aware* servers, which deliver the images in the best possible quality, considering network performance, server load, and a client-specified delivery time. The low-level content (e.g., color and texture) of each image in the repository is extracted to define feature vectors, which are organized in a database index at the search engine.

Physical separation of the image library index (in the search engine) from the image repository (in the server) facilitates distribution and mirroring of the library. The core components are connected by a coordination layer that isolates the details of network access and gives each component a maximum of flexibility to take advantage of future developments. Further details about the coordination layer and the overall Chariot architecture can be found in [193]. Chariot's indexing methods are described by Weber et al. [194]; the feature extraction methods for similarity assessment are detailed by Dimai [40].

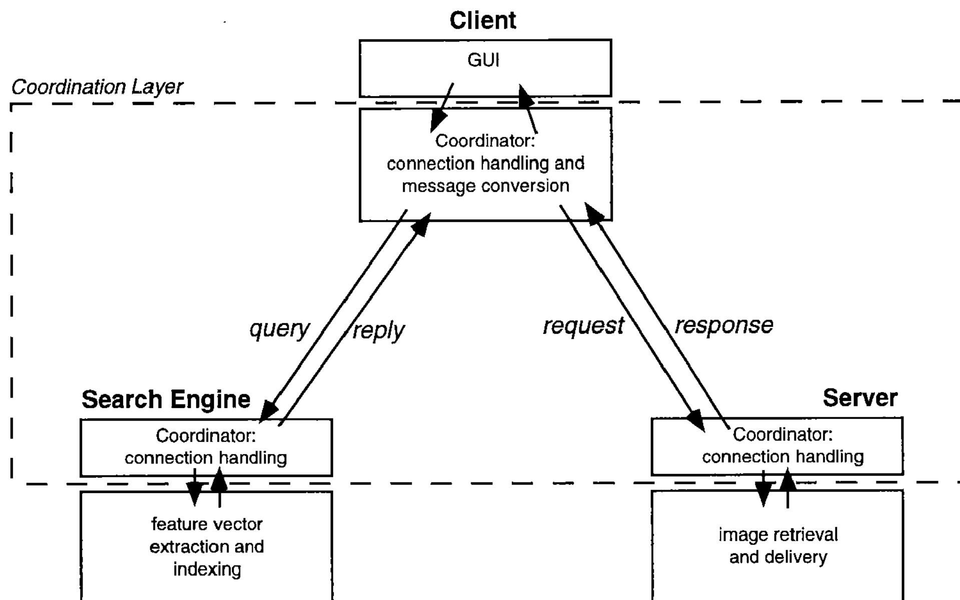


Figure 5.1: (Simplified) Chariot system architecture.

It is the adaptive image server that is relevant to our discussion of *network-awareness* and which serves as proof-of-concept for the ideas presented in this dissertation.

5.1.2 Instantiating the framework

The class diagram in Figure 4.2 shows that concrete implementations for the abstract classes *Type, Quality, Algorithm,* and *ResourceModel* must be provided to construct the network-aware image server. Furthermore, an application-specific *request parser* must be supplied (attribute of the class *RequestHandler*). The discussion of the issues involved in extending the framework refer to the class diagram depicted in Figure 4.2. The application-specific subclasses will not be shown because the entire class hierarchy should become obvious by the following description. The resource models are discussed in Section 5.1.3.

Type. Chariot deals exclusively with image data. Therefore, the abstract class *Type* is extended by an *ImageType* class. Chariot can support different specific image encodings, that is, image formats. The current implementation supports the two widely used GIF and JPEG formats. The subclasses of *Type* provide application- and media type-specific functionality. Types do not carry state, so that only one instance per *Type*-subclass is required (singleton pattern [65]). The factory method [65] *createQuality()* defined by *Type* and overridden by *ImageType* returns an instance of the *Quality*-subclass *ImageQuality* described next. Because the type "knows" how to interpret the quality attributes of its associated request entries, it lies in the responsibility of the type class (i.e., its method *versionMatch()*) to decide whether two versions of the same object (e.g., image) are of equivalent quality or not. For the same reason, and because the *Type* subclass maintains a list of applicable transformation algorithms, it is also well positioned to determine which algorithms must be employed to transform an existing version of an image to the targeted quality (method *algoChain()*). The algorithm subclasses then know which parameters must be used to produce the desired quality (see below).

Quality. The quality of an image (subclass *ImageQuality*) is defined by attributes *width, height, color depth, encoding*, and optional encoding-specific parameters. An example of such an encoding-specific parameter is the so-called JPEG quality factor provided by the JPEG format [77]. This quality factor determines the degree of lossy compression employed to encode a particular image and can be varied from 0 (lowest quality) to 100 (highest quality). The method *utility()* provides a quality metric as a function of the attributes that define image quality as explained in Section 3.3. Concrete implementations of the *utility()* method for image quality are discussed in Section 6.6.

Algorithms. Our Chariot prototype currently provides the following transcoding algorithms. For JPEG images, compression (i.e., re-encoding of a JPEG image with a lower quality factor), scaling (factors 1/2, 1/4, and 1/8), and progressive encoding are provided. For GIF images, scaling and color depth reduction are supported. Further options that are currently not exploited include a lossy encoding (i.e. compression) of GIF images, e.g., as achieved by *gifmunch* [59], or progressive encoding of GIF images as described by Amer et al. [5]. The supported algorithms are all registered with the single instance of the class *ImageType*. Each of the *Algorithm*-subclasses must implement the following methods. The method *targetQuality()* computes the resulting quality when applying the algorithm at a given parameter level to an image of a specified quality. *transformParam()* finds in the set of applicable parameters the parameter that is best suited to transform a given image version to the targeted output quality. *transformParam()* and *targetQuality()* are "inverse" functions and are both used in Program 4.5. The methods *isReducible()* and *isExpansible()* determine whether a given image quality leaves room for reduction or expansion with respect to the particular algorithm and the user-specified boundary conditions on minimal and maximal image quality, respectively. Finally, the method *apply()* transcodes an image at the given parameter level and produces a new version of the image (instance of class *Version*; Figure 4.2). The algorithms used in Chariot are all based on the programs provided with the NetPBM image processing toolkit [146]. The communication between different transcoding algorithms is file-based. (In the current implementation, intermediate versions of images produced by P_{prep} are written to disk.) Although such an operation may be wasteful in terms of time required for complex transformations (employing several algorithms in sequence), the approach has the benefit of supporting the caching of intermediate versions with little overhead.

RequestHandler. Because a server may have to serve multiple (network-aware) applications, we pursue the following approach. The application developer implements a prototype request handler to be registered with the server. Upon arrival of a new request for the particular application an appropriate handler can be cloned to process the request (prototype pattern [65]). Three issues must be addressed when building the prototype handler. First, the appropriate adaptation strategy must be chosen (Chariot uses the *Approximative* strategy described in Section 4.5.2). Second, the types supported by the application must be registered with the handler. (Chariot supports only images.) Third, an application-specific method to parse and process incoming requests must be supplied. Chariot's coordination layer uses a custom protocol [192]. A request message contains a list of objects to be delivered. Each requested object is characterized by the following information provided by the search engine: a *type* that allows instantiation

of the correct *Type*-subclass, a *local name* that uniquely identifies the requested image in the image repository, its *rank* within the request list, and *similarity* information. Rank and similarity are measures for the relevance of the image (with respect to the user's query image)[1]. Furthermore, a request message contains restrictions on the minimal quality tolerable and maximal quality beneficial to the user. This information is type-specific. For images, restrictions on image resolution and color depth can be specified. Moreover, the request message can limit the formats/encodings that can be handled by the client and specify whether reordered object delivery and/or progressive delivery are acceptable. Finally, the request contains the limit on the response time to be respected by the server.

5.1.3 Resource models

The model-based adaptation implemented by our framework relies on two application-specific resource models to characterize the work of the transcoding algorithms used by the application. The two models must provide estimates for the size of a transformed object and the CPU costs of a particular transformation. There are two issues to note:

Accuracy. These resource models need not be fully accurate. Unlike in the case of real-time systems, for example, where accurate estimates on the worst-case execution time of a task are critical to the reliability of the system, accuracy of both the cost and size models is less of an issue in adaptive systems. Inaccurate estimates may lead to suboptimal adaptation decisions (see Section 6.7), however, due to the repetitive nature of the adaptation process, the application may often be able to take corrective action later.

Complexity. Models for application resource usage may exhibit considerable complexity. Thus, it may not always be possible to find appropriate resource models for each transformation algorithm, or it may be too time-consuming (for an application developer) to find and validate suitable resource models before deploying the application [12]. Therefore, the goal from a software engineering perspective must be to provide a few reusable building blocks that can be composed and customized by the application developer to construct more complex resource models.

In the following, we first illustrate how non-trivial resource models are constructed in the case of Chariot and the lossy re-encoding of JPEG images (called JPEG compression). These models are used in the evaluation of our framework (Chapter 6). Second, we briefly introduce a few building blocks that facilitate the implementation of new resource models.

Chariot resource models

From the Chariot image repository, which contains more than 100'000 images, approximately 4'000 JPEG images of various JPEG quality factors and sizes have been chosen to experimentally derive suitable cost and size models for JPEG compression. Walther [189] finds that the output size of an image, that is, the size of an image after JPEG compression, depends on three

[1] In Chariot, the metric "rank" is a relative measure that orders the images. The metric "similarity" is an absolute measure that reflects the distance in the feature space between an image and the query image [191].

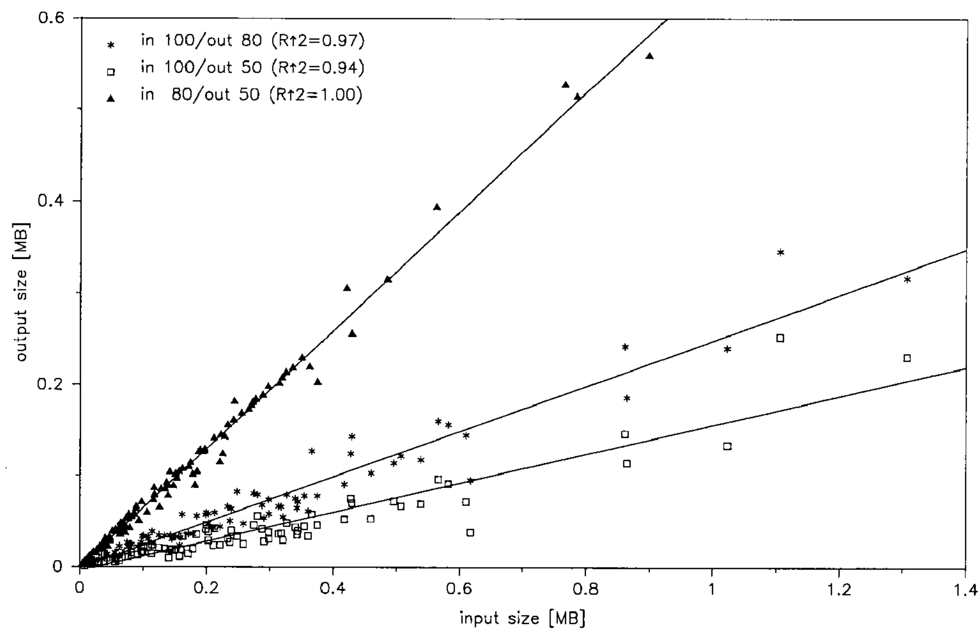


Figure 5.2: Sample data for a size model for JPEG compression. The figure shows how output size depends on input size for the following transcodings: (i) JPEG images with input quality factor 100, re-encoded with output quality factor 80 (star); (ii) input quality 100, output quality 50 (square); and (iii) input quality 80, output quality 50 (triangle). The lines represent a least square fit through the respective data points. R^2 denotes the coefficient of determination for the fit.

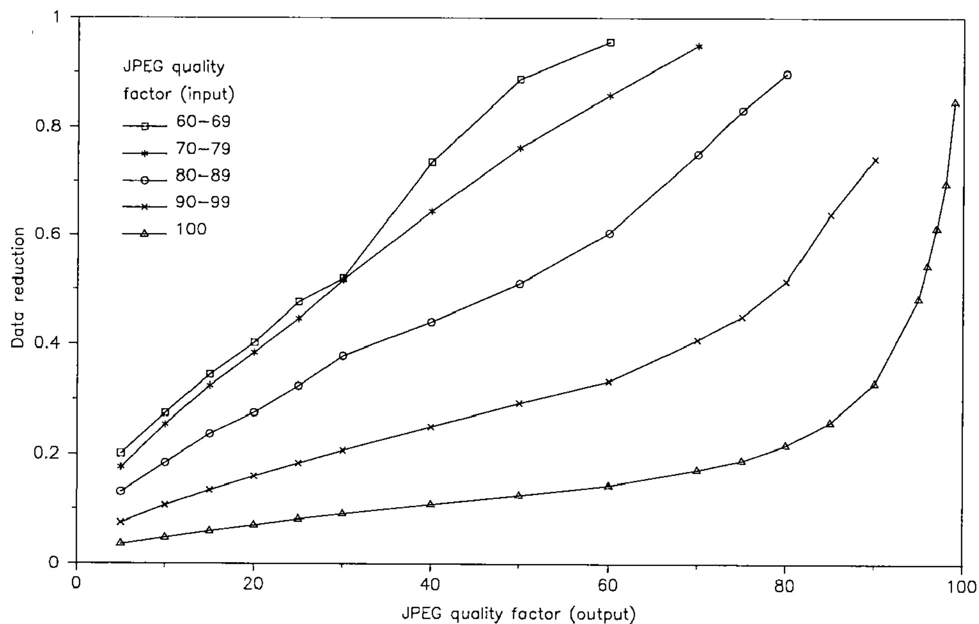


Figure 5.3: Size model for JPEG compression: median data reduction as a function of output quality factor. Medians are taken from images clustered according to their input quality factor.

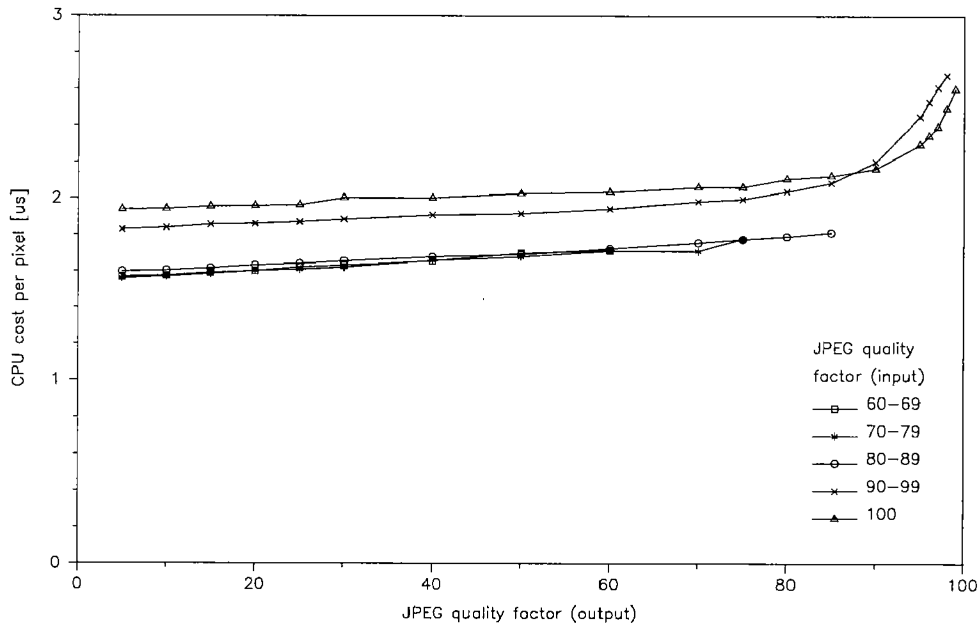


Figure 5.4: Cost model for JPEG compression: median CPU costs per pixel as a function of output quality factor. Medians are taken from images clustered according to their input quality factor. Transcoding costs are measured on a 200 MHz Pentium Pro PC running NetBSD 1.3 and are reported in microseconds.

factors: the input size, the input, and the output quality factor. For a particular combination of input and output quality factors the size of the transcoded image is a linear function of its input size. Figure 5.2 shows an example of these relationships. Since output size depends linearly on input size (for a particular combination of input and output quality factor), we can plot the size reduction, given by the ratio of output size and input size, as a function of the targeted output quality. Figure 5.3 shows the resource model for size reduction as a function of input and output size. Similar findings are reported by Chandra et al. [33][2].

Similarly, we find that CPU costs depend on the number of pixels, input and output quality factors (and the CPU power, of course). For a particular combination of input and output quality factors the costs are a linear function of the number of pixels. Figure 5.4 shows the resource model for the transcoding costs per pixel on a moderately fast PC. The CPU costs reflect the user and system time required for the transcodings. It is interesting to note that the costs per pixel do not seem to be effected by the input quality for quality factors smaller than 90 and that the costs seem to decrease (linearly) with increasing compression, that is, towards smaller output quality factors.

Other transformation algorithms employed by Chariot exhibit considerably simpler behavior in terms of size reduction and transcoding costs. E.g., the size reduction of the scaling algorithms is a square function of the scaling factor (or a linear function of the squared scaling factor).

[2]Chandra et al. [33] describe a fast method to determine the quality factor of a JPEG image by interpreting the information provided in the quantization tables stored with the JPEG image. Their algorithm would be a useful addition to the Chariot system.

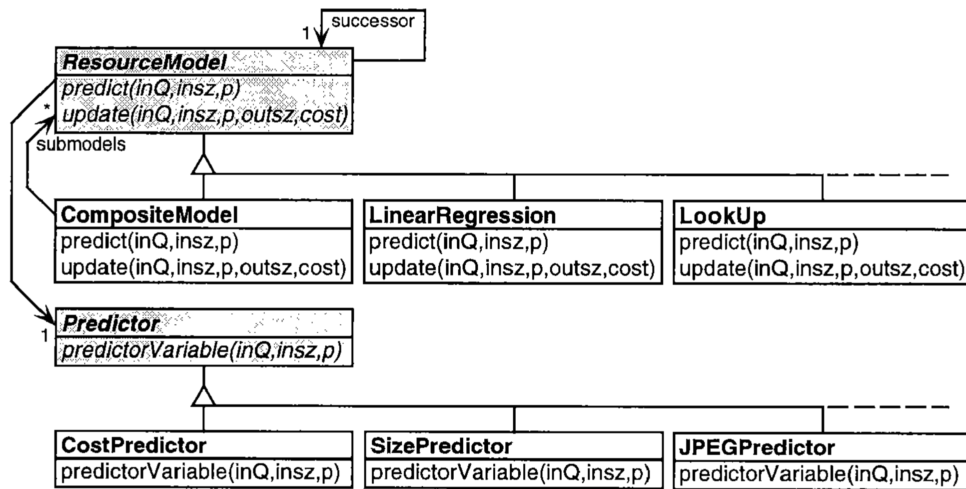


Figure 5.5: Class diagram for resource model building blocks (in UML-notation [56]).

Building blocks

Although the resource models for the transformation algorithms in Chariot seem fairly complex, complexity can often be reduced by composing the models from simpler models. E.g., the linear relationship between the input and the output size for the re-encoding of JPEG images (from a particular set of input and output quality factors) can be captured by linear regression models as shown in Figure 5.2. On the other hand, if all the images contained in the Chariot image repository had the same resolution (and hence approximately the same size), an even simpler model could be used. E.g., the size reduction could simply be looked up in a table indexed by the input and output quality factors. Based on these observations, our framework includes a few classes which can be used as building blocks to compose appropriate resource models. Figure 5.5 shows excerpts from the class hierarchy. The classes *LinearRegression* and *LookUp* provide the functionality described above. The class *CompositeModel* (composite pattern [65]) allows to aggregate multiple simpler (e.g., linear regression) models, e.g., for different ranges of input quality factors.

Additional flexibility is gained by decoupling the models from the predictor variables. E.g., for JPEG compression, we found that input size is the appropriate predictor variable for output size, but, transformation costs are better approximated as a function of the image resolution. For a particular combination of input and output quality factors both the size and the costs are a linear function of their predictor variable. Similar observations can be made for other transformation algorithms. Thus, decoupling the models from the predictor variables (subclasses of class *Predictor*) helps avoid a proliferation of resource model classes. A corresponding hierarchy of builder classes (builder pattern [65]; not shown in the figure) is used to initialize the resource models and to associate them with the appropriate predictors.

Because developing sophisticated resource models for a new network-aware application may often be (too) time-consuming (prior to deployment), the models (and their building blocks) should allow for on-line information collection of data about resource consumption by the various algorithms and should allow for dynamic refinement of the models at run-time. For this purpose, the resource model classes must implement an *update()* method. To illustrate the practicality of the update concept, we describe how a linear regression model could be updated

(without having to store all the past measurements). Let x be the predictor variable and b_0 and b_1 be the two parameters that define the regression line, then the response y is calculated as $y = b_1 x + b_0$. How to update the regression parameters b_0 and b_1 becomes obvious when considering how they can be computed (see [82]). The regression parameters that give minimum error variance are $b_1 = \frac{\sum_{i=1}^{n} x_i y_i + n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 + n(\bar{x})^2}$ and $b_0 = \bar{y} - b_1 \bar{x}$, where $\bar{x}$ and $\bar{y}$ denote the means of x_i and y_i respectively. By maintaining and updating the five variables $s_x = \sum x_i$, $s_y = \sum y_i$, $s_{xy} = \sum x_i y_i$, $s_{x^2} = \sum x_i^2$ and the number of measurements n, the regression parameters can easily be recomputed at any time.

Because it might not always be clear from the start which model is the most appropriate to characterize the size reduction achieved and the costs incurred by a transformation algorithm, it would be desirable if an application could use multiple resource models simultaneously, so that the most appropriate model can be selected after a trial deployment of the application. Furthermore, some models may only be applicable for certain ranges of predictor values. E.g., in Figure 5.4, the costs could be approximated by a linear function of the output quality for input quality factors smaller than 85, but would have to be approximated by a quadratic function for larger input quality factors. For these reasons, it would be useful to aggregate multiple resource models. These requirements are accounted for by the *successor* association among *ResourceModel* classes (see Figure 5.5), which reflects a chain of responsibility [65] among the different resource models. The *predict()* method call is forwarded along the chain until a model is found that can produce the estimate for the resource consumption sought. The *update()* call is forwarded to all the models in the chain, so that all (applicable) models can be updated for later comparison.

5.2 Potential for reuse

This section discusses the framework's reuse potential. First, we discuss how the framework can be extended to provide new functionality for Chariot-like applications (Section 5.2.1). Second, we illustrate how the framework could be integrated with a Web server to allow network-aware delivery for complex Web documents (Section 5.2.2). Third, we briefly discuss how the framework could be adapted to serve (slightly) different application domains (Section 5.2.3). Finally, limitations of the applicability of our framework are presented (Section 5.2.4).

5.2.1 Extending Chariot

During the design stages of the framework, emphasis has been put on extensibility. Options for framework extensions are listed in increasing order of complexity.

Algorithms. The framework can easily be extended to use new transformation algorithms for the image formats already supported (as discussed in Section 5.1). The developer must merely implement an appropriate *Algorithm*-subclass and the corresponding resource models (e.g., by composing some of the building blocks described) and register the class with the single instance of the *ImageType* class.

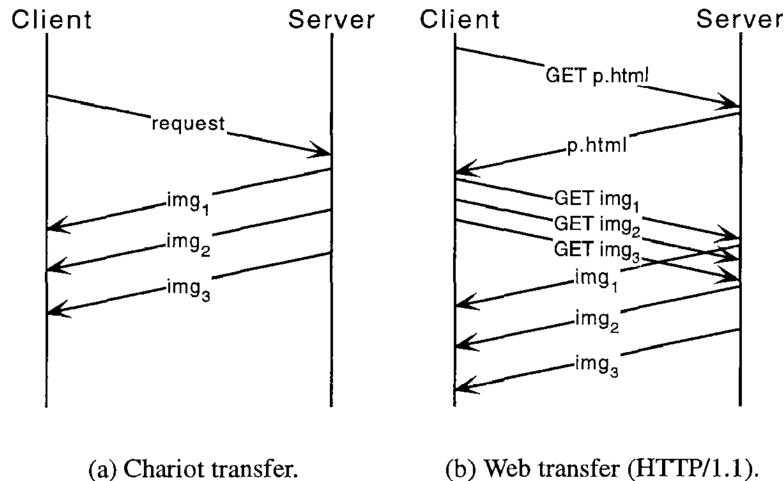
Encodings. New image encodings (or formats) can also be added at fairly small costs. We have not discussed how different formats are treated in the framework, but basically only a new format class with the supported transcoding algorithms must be supplied and registered with the *ImageType* instance. Furthermore, the method *algoChain()* may have to be adapted to select the appropriate encoding-specific algorithms. The rest of the framework is not affected.

Media types. Digital libraries may contain a variety of multimedia contents and hence retrieval services need not be restricted to searching and delivering image data. Some systems provide information retrieval capabilities on spoken documents (e.g., [195]). Other digital libraries, such as the one developed in the Informedia project [188], provide integrated video and audio retrieval techniques to search for relevant sequences in video data. For such systems, the response to a query may include a set of relevant video and/or audio clips. If these clips can be considered as individual entities (or objects) for delivery, then such multimedia retrieval engines could benefit from our framework for quality-aware adaptive content delivery, i.e., from timely delivery that respects the varying importance of the retrieved objects. Treating a video or audio clip as an entity means that play out at the client should not start before the entire video clip has been received, because inter-frame timings may not be preserved and hence cannot be guaranteed by our adaptive delivery process (see Section 5.2.4). To support such multimedia retrieval systems the framework must be extended by subclassing the abstract classes *Type* and *Quality* and by providing a set of applicable transformation algorithms for each of the new media types. The distillation dimensions are then implicitly defined by the transformation algorithms registered with the media type. A list of possible transcoding dimensions for common media types such as video and audio is supplied in Table 2.1. Otherwise, no changes to the framework are necessary, in particular, the adaptation process can be reused entirely. The Chariot protocol also allows extension of the system by new media types. Recall, that each requested object is associated with a *type* used to instantiate the appropriate *Type* subclass, so that objects with different types can be requested with a single request message.

5.2.2 Web object delivery

Web pages become increasingly complex. According to studies that date back three and more years, e.g., [23, 198], 50–75% of all the Web pages contained at least one image reference, the average number of images per page lying between 4 and 11 image references. Although there has been no similar study recently, we suspect that Web page complexity has increased considerably in the meantime. We use the term "Web object" to refer to complex Web pages consisting of an HTML [74] document that contains multiple references to embedded multimedia contents (images, audio, etc.). The size of such Web objects, that is, the aggregated size of the HTML page and all the embedded multimedia objects, is likely to have increased in step with the increasing number of image references found in Web pages. A common complaint heard from Web users is that downloading time is often unpredictably high [190].

Can our framework be used to make Web object delivery more predictable? The answer is yes, but before we can elaborate on how our framework can be integrated with a Web server, we must first overcome some fundamental differences between our service model (Section 3.2) and the way Web content is delivered (see Figure 5.6 for illustration). Web objects are delivered



(a) Chariot transfer.

(b) Web transfer (HTTP/1.1).

Figure 5.6: Differences in client-server interaction between Chariot and Web transfers.

as follows. The client requests the HTML page (*p.html* in Figure 5.6 (b)) from the server; upon reception, the client parses the page to extract inline references, and then requests the inline data separately (we use image data img_1, img_2, and img_3 for illustration purposes). This mode of operation can pose four problems. First, the client may choose not to request some of the inline images. Second, the client could delay the requests for the inline images by an arbitrary amount of time[3]. Third, not all images may reside on the same server. And fourth, the idempotency of such (GET-) requests and the stateless nature of Web servers make it difficult for the network-aware server to associate the image requests with the appropriate Web object. The first two issues are problematic, because they can entirely defeat the use of network-aware delivery to meet a user-specified time limit. For the remainder, we assume that all images are requested and that requests for embedded images are pipelined [50] and do not delay response delivery. If images must be fetched from other (not network-aware) servers, then the best the application can do is to try to control the delivery of those images that are co-located with the HTML page. The last aspect, associating requests with the corresponding request for a Web object, is addressed below by means of sessions.

Additional difficulties for network-aware Web content delivery comprise how to specify a time limit and how to specify relevance metrics (for individual parts of a Web object). Furthermore, a network-aware Web delivery mechanism should ideally have no (or only minimal) impact on how Web content is authored; and should not incur changes to the server or the browser.

We pursue the following approach to network-aware delivery of Web objects (see Figure 5.7). The user requests a Web object *p.html* from a server and wants the delivery to be bounded by 10 seconds, for instance. For this purpose, the browser (or the user) must rewrite the requested URL to include the time limit, e.g., as a query string *p.html?t=10*. The server

[3]Typically, either all embedded objects are requested or none. If no inline objects are requested, network-aware delivery may not be necessary at all. Second, no browser deliberately delays requests. Delays may occur due to differences in operation: A browser can choose between sequential, parallel and pipelined requests. Frystyk et al. [63] show that pipelining requests (HTTP/1.1 [50]) outperforms both sequential operation and multiple connections in parallel (popular with HTTP/1.0 [15]).

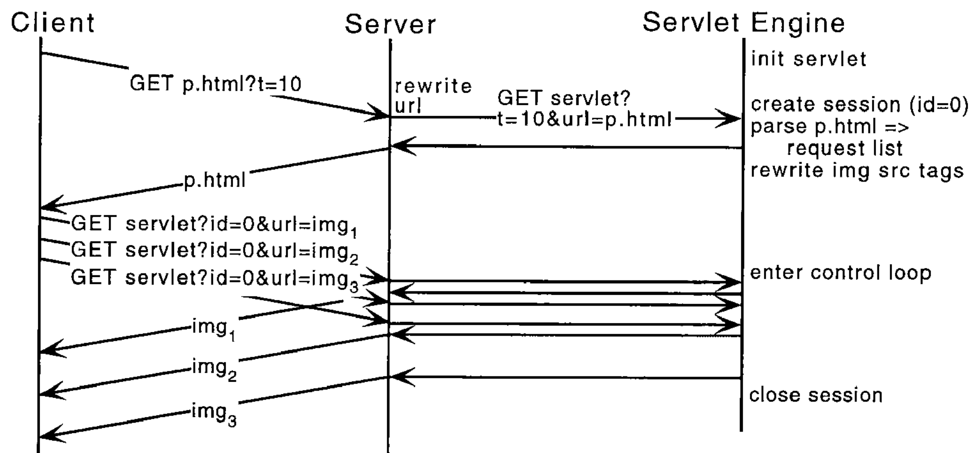
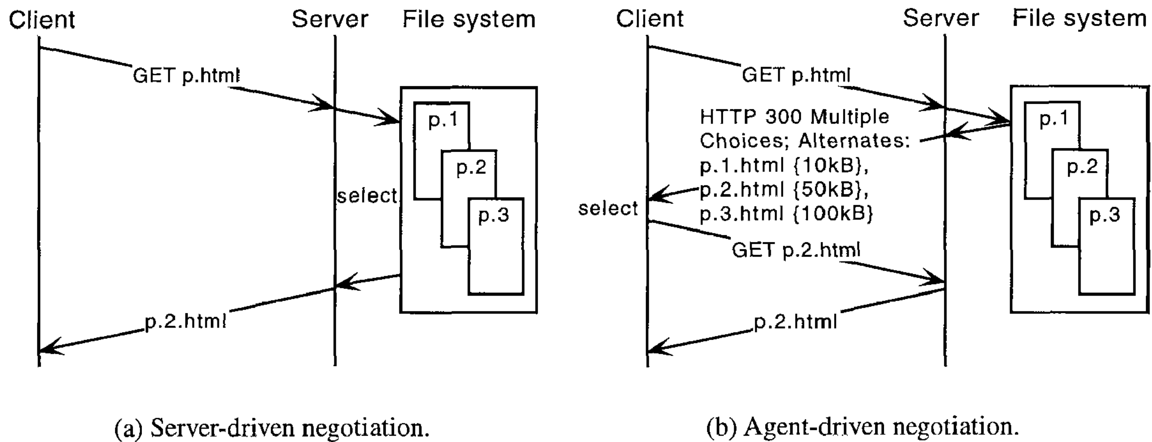


Figure 5.7: Network-aware Web content delivery (using Servlets).

then rewrites URLs of this format (e.g., by means of the *mod_rewrite* module in the Apache server [7]) to *servlet?url=p.html&t=10* and thereby forwards the request to the servlet engine (e.g., using Apache's *mod_jserv* module), where it is handled by a *servlet* [85] dedicated to network-aware content delivery[4]. The servlet creates a *session* for the particular request (schematically depicted by the session id 0 in Figure 5.7) and instantiates a new request handler, which is part of the framework. The request handler reads in the request (URL and time limit), retrieves the HTML page and parses it to extract information about the embedded objects (the image source tags in our example). This information is used to build the request list and to rewrite the image URLs to be sent back to the client with the HTML page. The image URLs are rewritten for two purposes: first, to forward the ensuing image requests directly to the servlet, and second, to be able to associate them with the requested Web object (by means of the session id). The control loop is entered and once the (pipelined) requests for the images arrive the entries in the request list are validated and the control loop can start to adapt the quality of the images, if need be, and to deliver them as described in the previous chapters. The framework extension implemented for Chariot can almost completely be reused for network-aware Web object delivery, because we are simply dealing with image data. Merely the request parser must be replaced to cope with the HTML format, and a mechanism to "validate" the request list must be provided.

Two issues concerning the service model for network-aware delivery have not been addressed so far. First, how to specify relevance metrics for individual parts of a Web object? There are two approaches. Either the content author specifies relevance scores, e.g., by means of special comments in the HTML-text (recognized by the request parser); or the client can try to guess the usefulness of the different images when parsing the HTML page. E.g., a client could guess which images reflect advertisements (using methods similar to those used in [87]) and assign appropriate relevance scores by extending the image URL by a query string,

[4]Instead of using servlets, FastCGI [46], a variant of the CGI [32] server-application interfacing, can be used. CGI defines application programs to be started by the Web server if a request for the program is received. After sending the response (via the server) to the client, the application terminates. This style of operation is wasteful because of the (repeated) application start-up overheads, and because this approach complicates the design of applications that need to conserve state between successive invocations. Servlets and FastCGI are two alternatives that remedy these problems.



(a) Server-driven negotiation.

(b) Agent-driven negotiation.

Figure 5.8: Server-driven versus agent-driven HTTP content negotiation.

e.g., as follows *servlet?id=0&url=*img_1*&relevance=r*. Second, to specify client capabilities, that is, boundary conditions on minimal and maximal quality, accept-headers as defined in the HTTP/1.1 specification [50] can be used.

To make Web content delivery network-aware as described in this section requires at most the following two changes. First, the way content is authored may have to be changed if relevance metrics are to be supported. Second, some means that allow a user to specify her preferences on response time must be provided. (This may be a static choice). An additional problem is how a client can learn whether a particular server supports network-aware content delivery or not—an issue that is beyond the scope of this dissertation (see [68, 185] for references on service location).

HTTP/1.1 content negotiation

How does the network-aware delivery of Web objects sketched above compare to the opportunities offered by HTTP content negotiation defined for HTTP/1.1 [50]?

Sometimes Web objects are available in alternate representations. For example, a text file may be available in several languages, or an image may be available in several sizes. HTTP content negotiation is a mechanism that allows client and server to select the most appropriate variant for a particular client. Two forms of content negotiation are distinguished in [50]: server-driven and agent-driven content negotiation. The type of content negotiation determines whether the server or the client is responsible to select a particular representation of a resource. Figure 5.8 illustrates the two HTTP content negotiation procedures.

It is important to note that HTTP content negotiation defines a protocol framework, but does not stipulate how this framework is to be used. Stemm [173] shows how agent-driven HTTP content negotiation can be used in conjunction with the SPAND network measurement architecture [163] to reduce response times of Web downloads (e.g., to a constant limit of 10 seconds). The client obtains a list of equivalent alternate versions of a page (see Figure 5.8 (b)). The client then retrieves a performance estimate for the server. Based on this performance prediction and the size of the variants, the client estimates the transfer time for the different versions of the Web page and chooses the one that most closely matches the user's requested response time.

For the client to estimate the transfer time for a Web page, it needs to know the combined size of the base HTML page and its embedded objects. To do this, Stemm adds a feature tag called "full-page length" to the base HTML page. Alternate versions of the HTML document have embedded images of different quality and hence different total page sizes. Server-driven negotiation can be used (i) to adapt to client heterogeneity, that is, to select a version of a document which matches the client's display or processing capabilities (specified by the accept headers in the request), or (ii) to manage the bandwidth consumption of busy Web servers [173].

How does the agent-driven approach compare to our type of network-aware content delivery? Points in favor of adaptive Web content delivery based on HTTP content negotiation are: first, decisions are made at the client, which implies that there is no need to know about server capabilities and thus no need for service location. Second, server-side overheads are considerably smaller (no URL rewriting, no servlets). On the down side we note that this type of content delivery only allows for a few static choices (the different variants available at the server) and that content cannot be served dynamically. This limits the adaptation flexibility and may make it difficult to match targets on time limit. Agent-driven content negotiation adds one round trip compared to normal Web delivery. Furthermore, the need to provide multiple versions of a page complicates the way content is authored and increases storage requirements. Most importantly, however, because adaptation decisions are made *before* a Web object is downloaded, decisions can only be made if bandwidth information from a client-side bandwidth cache (e.g., [163]) is available; and there is no way to cope with bandwidth fluctuations or wrong bandwidth estimates. Stemm [173] reports that up to 40% of all estimates produced by the SPAND system are off by a factor of 2 or more.

5.2.3 Other application domains

This section discusses how the framework could be adapted for other application domains.

Meet a reservation. As mentioned in Section 2.2, adaptation may even be required in reservation-capable networks. Applications may have to adapt to a confirmed reservation, which may be less than what the application has asked for in the reservation request. If the application is granted a given amount of bandwidth bw in the interval Δt, but the data it wants to send exceeds $\Delta t \cdot bw$, data reduction is necessary to meet the granted reservation. Can our framework help here? Of course, adaptation to meet a reservation is equivalent to adaptation to match network heterogeneity. The adaptation process can even be simplified. No bandwidth monitoring is required. P_{adapt} may not have to be invoked repeatedly, because bandwidth fluctuations are eliminated by the network. Dynamic adaptation may only be required if end-system resources, i.e., CPU power fluctuates considerably.

Meet a budget. Usage of network transport services may be charged in the future on a per-application, per-usage basis. There is an on-going debate over how charging and accounting should be implemented in the current (or in a future) Internet infrastructure (see [177] for an overview of research in this area). Some proposals argue that usage-based pricing is likely to be necessary to control congestion in the Internet [118]. By introducing the concept of "congestion pricing" these proposals try to build incentives into the network that restrain applications from

consuming large amounts of bandwidth in times of congestion [117, 64]. That is, if there is congestion in the network, an application is charged more for the same amount of bandwidth than it would have to pay for the same amount of bandwidth in times with excess capacity in the network.

Clearly, users want their spending to be bounded (that is, not to exceed a pre-defined budget) and they want reasonable quality for their money. Again, because network congestion is at best difficult to predict, applications are needed that can adapt their resource usage to meet their cost budget. An obvious approach (that minimizes costs) would be to transmit during periods when bandwidth is free, that is, in times with no contention for bandwidth. However, this may mean that the user has to wait indefinitely for an answer (if bandwidth contention prevails). Thus, we want the application to deliver the data both within a reasonable (user-specified) time frame T and within a cost budget. How can our framework be adapted to serve such applications? Because we concentrate on adaptation to a budget here, we assume for the sake of simplicity that the time limit is sufficiently coarse to transmit the original data (in an uncongested and therefore non-charged network), that is, we assume that the application needs not adapt to meet the response time. The idea is to replace the performance model developed in Section 4.4, which estimates response time as a function of resource availability, with a model that estimates cost as a function of congestion level (e.g., as reflected by the ratio of available and bottleneck bandwidth [143]) and pricing information. Given a model for the expected costs of a transfer, the control loop must determine at which bandwidth data can be sent for T seconds so that the budget is not exceeded. This means that appropriate quality reductions of the remaining objects in the request list must be found. The procedures outlined in Chapter 4 can be reused for this purpose. Instead of minimizing t_{diff} the control loop must try to minimize the difference between the money left and the estimated amount of money needed to transmit the remaining data. The transmit phase must be changed so that objects are "paced" into the network at the affordable rate.

Smooth object delivery. Rather than having predictable response times a user may wish that the various objects (e.g., images, image sequences, etc.) are delivered in a smooth fashion, that is, at a predictable rate. E.g., a Chariot user may wish to receive one image (approximately) every Δt seconds. How could our framework be adapted to allow for such delivery[5]? A straightforward solution employs hierarchical encoding schemes and progressive delivery as discussed in Section 4.8 and works as follows. If there are n images to be delivered, the time limit can be set to $T \approx n\Delta t$. The adaptation mechanisms described in Chapter 4 can be used to make sure that the size of the objects is reduced so that this time limit can approximately be met. Using hierarchical encoding for each of the objects to be delivered ensures that the (progressive) delivery for each of the objects can be stopped when Δt is reached. The quality reductions by the adaptation mechanisms ensure that the size of the objects is reduced sufficiently so that at least the first layer of each hierarchically encoded object can be delivered within Δt seconds.

This simple extension of our framework can be used, e.g., to support remote volume rendering applications such as the one described by Lippert et al. [98]. Their application hierarchically encodes volume data at the server and progressively transmits them to the client. If the application were to show a film of a "flight" through n 3D-objects, each encoded similarly, the

[5] We merely consider fairly coarse-grained adaptation here, and thus assume that $\Delta t > 1$ second.

technique outlined above could be used to enable a smooth stepping through frames, that is, the individual scenes of the "flight".

5.2.4 Limitations

Although the framework can be extended and adapted to serve as a foundation for a number of different network-aware applications, there are clearly limits to its applicability. The following list of limitations is by no means complete.

Delay-sensitive traffic. The framework is not suited to support adaptive delivery of delay-sensitive data, such as video or audio. Play-out of continuous media across a network requires much finer-grained control of the timing structures of individual video frames or talk spurts than the (relatively) coarse-grained adaptation mechanisms deployed in the framework can achieve. There are many adaptive video and audio applications that successfully adapt to bandwidth fluctuations and that are able to conserve inter-frame spacings at the fine granularity required for these applications, e.g., vat [81], vic [112], ivs [183], WaveVideo [45], or MTP [72].

Multicast transmission. The framework's sender-driven adaptation process described in Chapter 4 is clearly not a wise choice for multicast content delivery. Feedback implosion and the fact that adaptation must meet the response time target for the slowest client are only two of the problems such a scheme would be faced with. Network-aware content delivery in multicast scenarios is preferably achieved by intermediary nodes (e.g., providing active services [6]) in the network (see Keller et al. [90]), or in a receiver-driven fashion as pioneered by McCanne et al. [113]. We leave it to future work to study how our framework could be integrated with active services deployed in the network to provide the type of network-aware content delivery proposed in this dissertation to multiple receivers in a multicast session simultaneously.

Server selection. A fairly popular form of adaptation to achieve predictable response times is to pick (among a set of mirror servers) the server that is best suited to deliver the requested data. A mirror server is well suited if it can deliver the data in high quality within the desired time frame. As is, our server-side framework cannot easily be extended to support server selection. We note, however, that dynamic server selection is an orthogonal issue to the type of adaptation captured by our framework, and would therefore be a useful addition of applications derived from the framework. Such a scheme would preferably be deployed on the client side. Stemm [173], Crovella et al. [28], and Fei et al. [47] present different techniques for server selection. The former two make entirely client-based selection decisions. Myers et al. [126] discuss implications of Internet dynamics on server selection mechanisms.

5.3 Summary

This chapter explains how the framework is instantiated by presenting a sample application—Chariot, an image search and retrieval system —, that is derived from the framework. The

models that characterize application (i.e. Chariot) performance and that are necessary for the application-to-network quality mapping are presented and simple building blocks are discussed that facilitate composition of resource models for new transcoding algorithms (e.g., for new applications). The chapter discusses the potential of framework reuse in different settings. Chariot-like applications, i.e. other multimedia retrieval systems can easily benefit from the abstractions provided by the framework. Other applications, such as the transfer of complex Web documents containing a number of embedded objects, can also be supported by the framework with reasonable effort. The discussion of the framework's applicability to slightly different domains indicates that the framework should be flexible enough to allow for a rather painless transition to other application scenarios.

The fact that in all the application scenarios discussed, the framework core, i.e., its adaptation process, did not have to be revised, permits us to conclude the following. The framework-based approach to the development of network-aware applications is beneficial as it allows for reuse of the core adaptation process and can therefore shield developers from many of the complexities in dealing with network dynamics.

Chapter 6

Evaluation

In this chapter we want to answer such basic questions as (i) "does adaptation work?", (ii) "is it beneficial?", and (iii) "at what costs does such a (presumed) benefit come?". Previous work [130, 132, 59] had provided answers to these questions (adaptation *does* work, *can be* beneficial, and the benefits are obtainable at *considerably small* costs) and has thereby shown that the concept of adaptation can be worthwhile. In addition to these previous studies, our work goes a step further and establishes a detailed understanding of the complexities of the adaptive systems under consideration by means of a systematic evaluation methodology. We address questions such as, "what are the key factors effecting the performance of a network-aware system?", "How does the presumed benefit depend on the user's notion of utility?", and "What are the factors that primarily drive the costs of dynamic adaptation?"

When does adaptation work?

Even though network-aware delivery can provide acceptable application performance in a much wider range of situations than static solutions, adaptation clearly cannot solve all the problems related to variable and unpredictable network dynamics. To better understand the complex dynamic adaptation behavior, we aim at precisely characterizing the situations in which adaptation is able to fulfill its goals. Therefore, we must identify the key factors that effect our framework's ability to adapt. For the purpose of identifying the key factors, we distinguish between framework-*external* and *-internal* factors. External factors comprise network, end-system, application and request properties. Internal factors reflect aspects such as the accuracy of the resource estimators or the adaptation policies employed.

Determining the effect of the various factors on system performance allows us to draw conclusions with respect to *when* adaptation works. E.g., we want to establish for what bandwidth and host load levels is a particular request fulfillable within a given time limit. Such findings provide the basis to tackle the second question, whether (and when) adaptation is beneficial. If adaptation works only for a very limited subset of bandwidth and load levels, it might not be useful at all.

Identifying the key internal factors and their effects on performance helps to find answers to questions such as: "How sensitive is model-based adaptation to the accuracy of the models employed"; and in particular, "how important is the accuracy and timeliness of the bandwidth estimates?".

Is adaptation beneficial?

The answer to whether adaptation provides any benefit to a user (compared to a static policy for content delivery) mainly depends on two issues. First, it depends on how much the user values a predictable response time. Second, the answer is determined by the user's notion of quality.

Since these two measures are highly application- *and* user-dependent, there is clearly no final answer to whether adaptation is beneficial or not. Even in the context of a particular application, we cannot assess a potential benefit without a well-founded user-study. Because it is not our goal to understand all the particularities of a single application and its user community, we do not try to tackle the general question whether adaptation is beneficial, but rather concentrate on evaluating under which circumstances adaptation can provide a quantifiable benefit.

How much does adaptation cost?

To better gauge the benefit of adaptation, we must relate it to the costs incurred by the adaptation process. In addition to the hardly quantifiable costs for the development of network-aware applications (see Chapter 5), the run-time costs for dynamic adaptation must be taken into account. There are basically two sources of overhead: the decision overhead incurred during the adapt phases, and the CPU resources needed for the bandwidth tradeoff. Again, we are interested in quantifying the effects of various external and internal factors on the adaptation costs.

The remainder of this chapter is organized as follows: Section 6.1 describes the evaluation methodology used. Sections 6.2–6.4 and 6.7 address question (i), that is, whether and when does adaptation work? Section 6.2 presents examples to illustrate that the model-based adaptation proposed in this thesis can fulfill its objectives. Sections 6.3 and 6.4 systematically identify and evaluate the key *external* factors and their effect on performance. Before studying the utility of adaptation, Section 6.5 quantifies the adaptation overhead as a function of the performance-relevant factors identified in Section 6.3 (and thereby answers question (iii)). Addressing question (ii), Section 6.6 illustrates that the benefit of network-awareness heavily depends on the user's notion of quality (and hence on the choice of utility functions). Resuming question (i), Section 6.7 finally discusses the importance of accurate resource models (and in particular of accurate bandwidth estimation) for network-aware delivery.

6.1 Evaluation methodology

This section defines the performance metrics and factors used for the evaluations in the following sections. It also describes (and briefly evaluates) trace modulation [133], the technique used to ensure the reproducibility of the experiments.

6.1.1 Performance metrics

From a user's point of view, there are basically two aspects that characterize the performance of an adaptive application adhering to the service model defined in Section 3.2:

Time. The application's ability to deliver the requested objects within the user-specified time limit.

Quality. The application's ability to maximize the quality of the objects delivered.

The first aspect can be quite easily captured by the deviation from the time limit. Characterizing the second aspect is less straightforward; the problems that arise when trying to quantify it are twofold:

- How to quantify the *quality* of the response?
- How to assess the system's ability to *maximize* the quality of the response?

In our framework, the overall quality of a response is quantified by utility functions as introduced in Section 3.3. There are two drawbacks to the use of utility functions for the evaluation of a framework for adaptive applications.

First, utility functions are application-specific and the results may not be of much use in a more general setting. Second, even in the context of a particular application, e.g., Chariot, such an approach would be at least questionable without a user study that tries to come up with realistic, well-founded utility functions for the application. Since it is not the main focus of our study to produce such utility functions (e.g., for Chariot), we defer discussion of the impact of different utility functions on adaptation behavior to Section 6.6 and use a more application-independent metric: we approximate the quality of a response with the amount of (application-relevant) data delivered.

Second, provided that we are able to quantify the quality of a response, how do we assess the system's ability to maximize it? That is, for a particular outcome of an experiment, how can we know how close we are to the maximal quality theoretically achievable? To compute the maximal quality theoretically achievable, we must know about the resources available to the application. Unless we are in full control of the application's environment and in particular the resources (network and CPU bandwidth) available to the application, it is at least a daunting task to try to infer the resources available from observations of an application run. For example, if we were to observe the transfer of images from a Chariot server to a Chariot client over the Internet, which is clearly beyond our control, and if we found the server does not transmit images at some point in time, e.g., because it is busy making adaptation decisions, we cannot know how much bandwidth would have been available to a more sophisticated Chariot server able to make decisions without incurring idle transmission periods. However, if we are in full control of the resources available to the application, we can assess the amount of data theoretically transmittable within the time allotted by the user and are therefore in a position to compute the maximal quality theoretically achievable. As a consequence, we conduct our experiments in a controlled environment.

In our experiment, we exert control over (i) the CPU resources, and (ii) the network resources available to the application by using (i) a dedicated, otherwise idle host for the network-aware sender, and (ii) by transparently emulating the bandwidth available to the application over a high-speed LAN using a technique called *trace modulation* [133]. Section 6.1.4 details how bandwidth emulation is performed for our experiments.

Bandwidth emulation in a controlled environment accomplishes two objectives: we can reproduce the experiments, and it enables us to compute the data effectively transmittable within

a specified time frame. This second property provides a simple mechanism to approximate and quantify the second performance metric (quality) as the ratio of transmitted data and transmittable data (or the *utilization* of the bandwidth available to the application).

6.1.2 External factors

The following two sections try to "span the factor space", that is, the sections try to qualitatively identify the factors that presumably have a non-negligible effect on the performance. This section covers framework-*external* factors. The factor space can roughly be divided into three domains: network conditions, end-system conditions, and application parameters. These domains are subdivided further as follows:

Network conditions

There are two major factors related to network conditions that may influence application performance:

Bandwidth. The higher the bandwidth is, the smaller the time frame for "prepare activities" is, because the transmit phases finish quickly. Thus, it is more likely that we experience idle transmit periods and hence low bandwidth utilization.

Volatility. Frequent oscillations (or a high *volatility*) of the available bandwidth require frequent adaptation decisions. Frequent adaptations incur high decision-making overheads and may also increase the risk of making wrong adaptation decisions.

End-system conditions

Among the countless parameters that effect the performance of a computer system (for a given workload), we merely consider the following two factors[1]:

CPU. The factor CPU expresses the *CPU bandwidth* in terms of instructions per second.

Load. The factor *host load* indicates how many processes share the particular CPU bandwidth.

Application parameters

There are several factors that could have an effect on the behavior of the adaptation process:

Request. The two factors that are used to characterize a request for our performance evaluations are: (i) the number of objects n to be delivered, and (ii) the distribution of the object sizes within a request, e.g., constant size objects, uniformly distributed, etc. Both factors—in addition to the adaptation policy employed—directly influence *agility*, i.e., how fast the application can react to changes in the (network) resource availability. For

[1] Clearly, there are other end-system related factors, such as memory or cache size, that may effect performance. However, we expect these two factors (CPU and load) to have the most pronounced effect on the time (t_{prep}) used to transform an object. These two factors are also accounted for in our adaptation models (see Section 4.4).

a simple adaptation policy, such as described in Section 6.1.3, the following holds: the more objects there are in a request, the more adaptation points there are and the more agile the application behaves. Likewise, constant-sized objects can positively effect agility, as they produce an equally spaced series of adaptation points.

Adaptation potential. The adaptation potential of the objects requested depends on the "original size" of the objects, that is, d_{left} at the start of request processing, and the size of the objects when reduced to the lower limit on object quality. The higher the adaptation potential is, the better the chances are that the application can react to large changes in resource availability.

Adaptation granularity. The number of transformation algorithms and the number of parameter levels per transformation applicable to each of the objects in a request define the *granularity of adaptation* possible. The more parameter levels are applicable for a transformation, the finer the adaptation granularity and the better the chance that quality reductions can be distributed evenly among the objects in a request. Hence, a fine adaptation granularity may facilitate attaining a high utility. On the other hand, it may adversely effect adaptation costs, as each transformation may introduce some fixed overhead. Note, similar arguments also apply to the factor *adaptation potential.*

Adaptation required. The *degree of adaptation required* can be described by the ratio of the data to be transferred ($d_{left}(0)$) and the data effectively transmittable in the time frame allotted by the user. (The degree of adaptation required should be smaller than the adaptation potential, otherwise the application is bound to miss the time limit.) We would expect that the higher the degree of adaptation required, the harder the application must work to meet the time limit and hence the higher the probability that the deadline is missed.

This definition of *adaptation required* is problematic for the design of our experiment, because (i) not every ratio between 1 and the adaptation potential may actually be achievable, and (ii) it does not quite reflect "how hard" the application must work to meet the time limit.

First, recall from Section 4.5 that for a request with n objects and an average adaptation granularity of m transformation steps per object, there are $n_r \approx m^n$ possible responses. Each of these responses may require a different amount of data to be transmitted. Even though n_r may be fairly large, it is still finite. As a consequence, the distribution of response sizes (and hence achievable reduction ratios) is discrete and there may be ratios that are not achievable with the given request and adaptation granularity. The distribution of response sizes, if non-uniform, gives rise to the second concern: We suspect that the fewer possible responses there are for a given amount of data transferable (and hence for a given degree of adaptation required), the less flexibility there is to make adaptation decisions and the more likely it is that deadlines will be missed.

Therefore, the notion of *adaptation required* should capture both aspects: how much data reduction must be achieved and how likely it is that a given reduction can be achieved (see Section 6.3.2 for details).

It may be worth noting that the user-specified *time limit* is not included in the list of factors. The time limit is not a free variable, that is, an independent factor, because it can be derived from network conditions, request properties, and the degree of adaptation required.

6.1.3 Internal factors

Accuracy of resource models

An important factor that may effect the performance of model-based adaptation is the *accuracy* of the resource models employed, that is, the accuracy of both bandwidth and load estimation, as well as the accuracy of the models for resource consumption. The accuracy of the various estimators is a concern that is orthogonal to the external factors listed above and is therefore treated separately. Section 6.7 discusses how the accuracy of resource estimation affects performance.

In the context of the experiments conducted in Sections 6.2 and 6.3 we attempt to employ simple, but "reasonably accurate" resource models. The requirement that the simple resource models used are "reasonably accurate" is satisfied by the choice of experimental setup: The bandwidth estimator reports instantaneous samples of the bandwidth currently available to the sender, that is, due to the use of trace modulation (Section 6.1.4), it reports the bandwidth that is currently being emulated. The estimator is simple as it does not consider any history to predict the bandwidth available for t_{left} and just assumes that the bandwidth will stay constant for this period. It is reasonably accurate, because it accurately reports the current bandwidth. The load estimator used is based on the UNIX standard load measurement technique, which approximates the end-system load by averaging the size of the scheduler run-queue over an interval of n seconds (typically $n = 60$) [119]. This may not be the most accurate indicator of instantaneous system load, however, since the experiments are conducted in an otherwise unloaded system the end-system load induced by daemon processes etc. is fairly small ($\ll 1$) and thus the prepare time t_{prep} is dominated by the prepare costs c_{prep}, that is, $load \approx 1$ (self-induced load). To minimize effects of inaccurate estimates for the resource consumption of the transformation algorithms used in the experiments, the models for CPU usage and reduction achievable by a given transformation are precomputed for all the objects requested in the experiments (see Section 5.1.3).

Adaptation policy

A second important internal factor that may effect the performance of model-based adaptation is the adaptation policy employed by the control loop. The adaptation policy determines the notion of relevance of individual objects in a request, and the order of object delivery, for example. The adaptation policy used in the experiments in Sections 6.2 and 6.3 is kept as simple as possible to reduce the complexity of the initial evaluation. The adaptation policy used assumes that all the images in a request are equally relevant. This assumption implies that the control loop tries to distribute the reductions evenly among the images in the request. The objects are delivered in the order requested, that is, the adaptation policy does not try to be smart and reorder the images in an attempt to optimize the alignment of prepare and transmit phases for instance. Therefore, when interpreting the results presented in the following sections, we must keep in mind that we basically analyze a "worst case" scenario. Any even slightly more sophisticated

adaptation policy can be expected to improve the adaptation performance as indicated by the results presented in Section 6.3.3. More sophisticated adaptation policies, e.g., using different notions of relevance are studied in Section 6.6.

6.1.4 Bandwidth emulation

Since ensuring reliable and reproducible experiments on real networks is extremely difficult, we follow the approach of other researchers and resort to a technique called *trace modulation* [133]. Trace modulation performs an application-transparent emulation of a slower target network, e.g., a wide-area network such as the Internet or a wireless network, on a faster, wired local-area network. Each application's network traffic is delayed according to the bandwidth parameters read from a so-called *replay trace*, which is gathered from monitored transfers, or which can be produced synthetically to study particularly interesting phenomena.

Since a study of the interaction of multiple network-aware flows sharing a bottleneck link is not the primary focus of our work[2], we perform the bandwidth emulation at the transport level and not at the network level as in the case of Noble's implementation [133].

Trace modulation is implemented by means of sender-based traffic shaping. We use a variation of a *token bucket* traffic shaper whose tokens (or credits) are updated at a variable rate (according to the bandwidth replay trace modulated). This is in contrast to the the "classic" token bucket scheme [139], where tokens are updated at a fixed rate. Our implementation employs a *use-it-or-lose-it* strategy, that is, the credits C are only valid for a given time interval ΔT and are reclaimed if unused. The application can use up to C credits, i.e. can send up to C bytes, within the interval ΔT. If the application has $D > C$ bytes to send, it can only send C bytes within the first interval ΔT and has to wait for the bucket to be replenished, that is, it must wait until the end of the interval ΔT before sending the remaining $D - C$ bytes. If the application consumes $D < C$ credits within ΔT, the remaining $C - D$ credits are lost. Unused credits are removed from the token bucket at the end of every interval and replaced by the credits available in the next interval.

Figure 6.1 illustrates that the trace modulation technique sketched above can fairly accurately emulate a given bandwidth trace. The plot shows the bandwidth trace to be emulated (solid line). The trace was obtained by monitoring an Internet transfer (the same trace will be used in an experiment in Section 6.2). For each time t the curve "emulated bandwidth" reports the bandwidth available to (and consumed by) a ftp-type application at time t, where the bandwidth is averaged over 20 runs of the application. The error bars depict the standard deviation of the bandwidth emulated. The figure indicates that the bandwidth emulated closely matches the replay trace and that variability is fairly low — a key to ensuring reproducibility of the experiments to be conducted.

[2]Recall that we expect the network-aware sender to be well-connected to the Internet and we expect that the bandwidth of the connections requiring adaptation is constrained downstream.

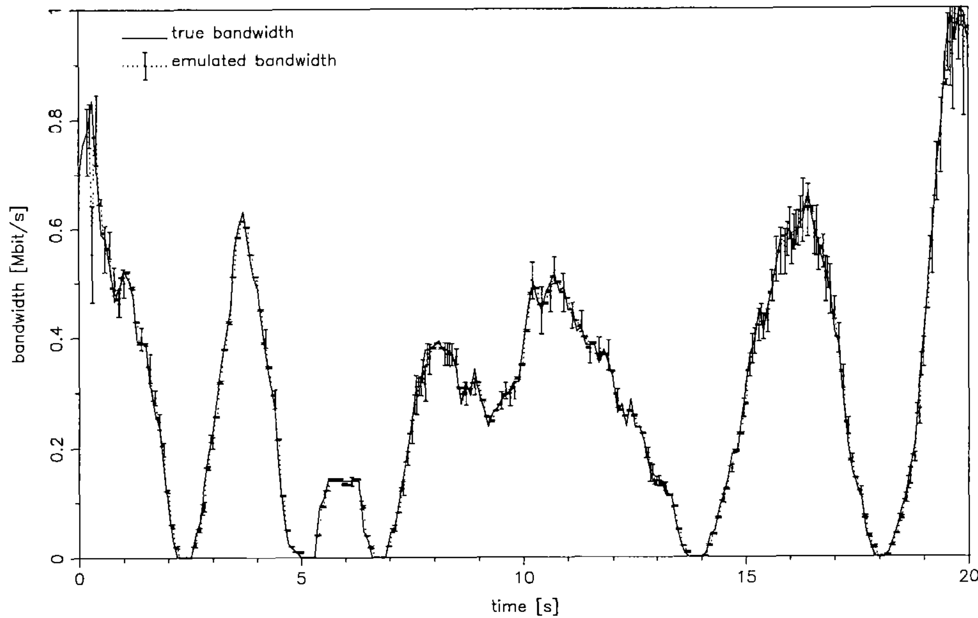


Figure 6.1: Bandwidth emulation — average and confidence interval for mean at 95% confidence level (20 runs).

6.1.5 Limitations

Application

Since the framework for model-based network-aware applications proposed in Section 4 is a white-box framework [86] and thus does not work "out of the box", it must be instantiated to form a concrete application that can be used in a performance evaluation. In the following sections, the Chariot image retrieval system, as described in Chapter 5, will serve as the platform for the experiments. Although not validated by experimentation, we see no reason why many of the "bottom line" results could not be carried over to other instantiations of the framework in particular and other types of model-based network-aware applications in general.

Steady-state adaptation

We separate the intrinsics of the start-up behavior from those of the adaptation mechanisms and draw a line between the two phases *start-up* and *steady-state* to shield the discussion of the adaptation mechanisms present in network-aware applications from the additional complexity incurred by approaches addressing the start-up problem (see Section 4.7).

Steady-state. The application (i.e., its control loop) enters the steady-state phase once it obtains (reliable) bandwidth estimates from lower layers and once the two stages of the prepare-transmit pipeline are filled. The goal of the steady-state adaptation mechanisms is to maximize the performance as characterized by the metrics described in Section 6.1.1.

Start-up. The start-up phase is defined implicitly by the period between the end of request analysis and the time (reliable) bandwidth estimates become available. The goals of a start-up policy are (i) to shorten this interval as much as possible, and (ii) to minimize

the risk that the time limit is exceeded, e.g. by an excessively long transfer of the first object(s).

Note that the goals of the two phases are orthogonal. This observation allows us to decouple the evaluation of the steady-state adaptation mechanisms from the evaluation of the approaches suggested to improve start-up behavior. This chapter concentrates on the performance aspects of steady-state adaptation behavior. The effectiveness of the various techniques addressing the start-up problem has been evaluated by Berger [13]. The results confirm the expectations expressed in Section 4.7 and thus will not be discussed here.

6.2 Does adaptation work?

Before turning to a systematic and detailed evaluation, we ought to get some intuition about how the basic adaptation mechanisms work in practice. In the following sections, the Chariot image retrieval system will serve as the platform for the experiments.

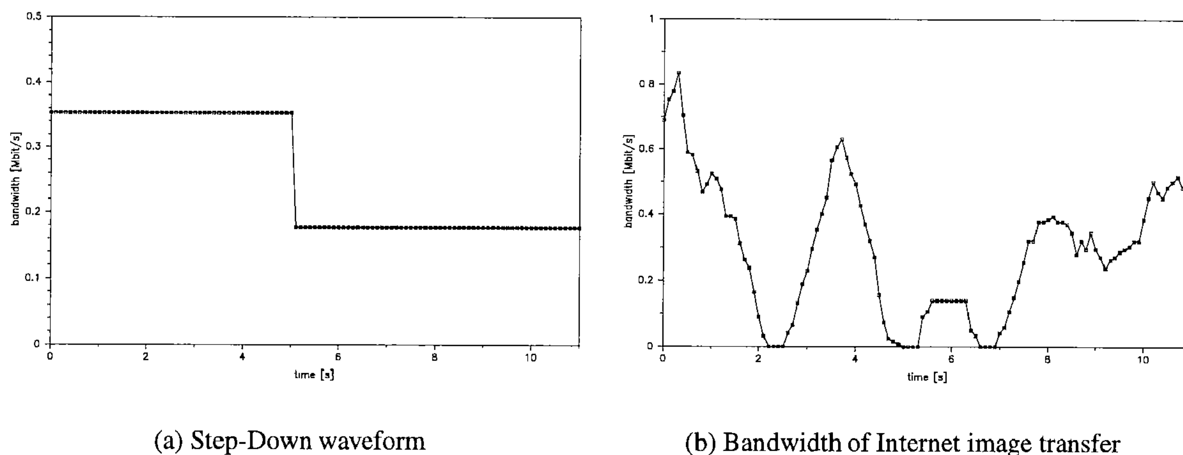
Our approach to illustrate the system's network-awareness proceeds in two steps: First, we subject the system to synthetic reference bandwidth waveforms. The example presented here is the *Step-Down* waveform shown in Figure 6.2(a) to characterize its ability to adapt in general and in accordance with the well-established principles for measuring dynamic response from the field of *control systems* [154]. Second, field tests in the Internet with its high bandwidth dynamics enable us to assess the system's agility with respect to real-world network traffic.

6.2.1 Experiment

In our experiments the Chariot server runs on a 200 MHz Pentium Pro PC with 64 MB RAM running NetBSD 1.3. A 134 MHz MIPS R4600 SGI Indy with 64 MB of memory serves as the platform for the client. For both of the experiments shown below, the client requests transmission of 25 JPEG images stored at the server in a resolution of 320×240 pixels and a JPEG quality factor of 100.[3] The 25 images total 1.03 MB of data to be transmitted. The images are assumed to be equally relevant, which means that equal relevance scores are assigned to the 25 images. The user-imposed time limit for request processing is arbitrarily chosen to be 10 seconds with a tolerance interval of $[-0.2, 0.2]$ seconds (or 2%). The tolerance interval limits oscillations of the adaptation process (see Section 4.5). Note that a 2% tolerance on the time limit is a fairly aggressive performance goal. This small tolerance has been chosen merely to challenge our implementation of the feedback loop and to see whether network-aware delivery is (at least in principle) capable of satisfying even ambitious performance goals. Often applications (and users) can tolerate response times that deviate more from the time limit. In fact, we will relax the tolerance in later sections and consider deviations up to 10–15% as being acceptable.

The bandwidth replay traces used for the two experiments conducted are depicted in Figure 6.2. The *Step-Down* waveform of Figure 6.2(a) is an idealization of real network scenarios; it approximates possible situations in an overlay network [89] for instance, where a mobile

[3]The query image and results are the same as in [178] (Figure 2). We use color features, enhanced with spatial information, for the similarity search [178, 194].



(a) Step-Down waveform

(b) Bandwidth of Internet image transfer

Figure 6.2: Bandwidth replay traces used.

client may seamlessly switch between different network interfaces. Figure 6.2(b) shows the monitor layer's perception of the available bandwidth during a transfer between ETH Zürich (Switzerland) and the University of Linz (Austria). The bandwidth curve was obtained using a moving average with a 1-second averaging interval. A new bandwidth estimate can be obtained from the monitor every 0.1 seconds. The periods of zero bandwidth reflect TCP timeouts. The step-down bandwidth trace was constructed, such that the same amount of data can be transmitted within 10 seconds as for the Internet trace. Therefore, for both scenarios, a reduction from 1.03 MB to 321.4 kB is necessary to meet the time limit.

Chariot's reduction algorithms registered with the framework are image compression (with various quality factors (99–95, 90, 85, 80, 70, 60, 50, 40, 30, 20, 10)) and resolution scaling (with a factor 1/2). The boundary on minimal quality is set to JPEG quality factor 10 and a resolution of 120 × 80 pixels. In an attempt to reduce the complexity for the illustrative examples, we run the experiments on an unloaded system and precompute the CPU resource consumption for the request and the transformations used. The server performs communication latency hiding by means of a separate thread for P_{prep}. As a consequence, P_{trans} for image i of the sequentially processed request list operates concurrently to P_{prep} for image $i+1$.

Each of the two experiments is repeated 50 times. The examples presented in the following depict the "median outcome" of the experiments, that is, the experiments presented have a median deviation from the time limit *and* a median bandwidth utilization. The medians for the two performance metrics for all of the 50 experiments are reported in Table 6.1.

6.2.2 Results

Step-Down waveform

Figure 6.3—a data vs. time plot as introduced by Jacobson [79]—shows that Chariot is able to both adapt the amount of data transmitted (curve named "actual") to the amount of data transmittable ("possible") and deliver the 25 images within the 10-second time limit. The time limit is exceeded by 1.7% (0.17 seconds), which is within the 0.2-second tolerance. The *Step-Down* waveform of the available bandwidth in Figure 6.2(a) represents the derivative of the

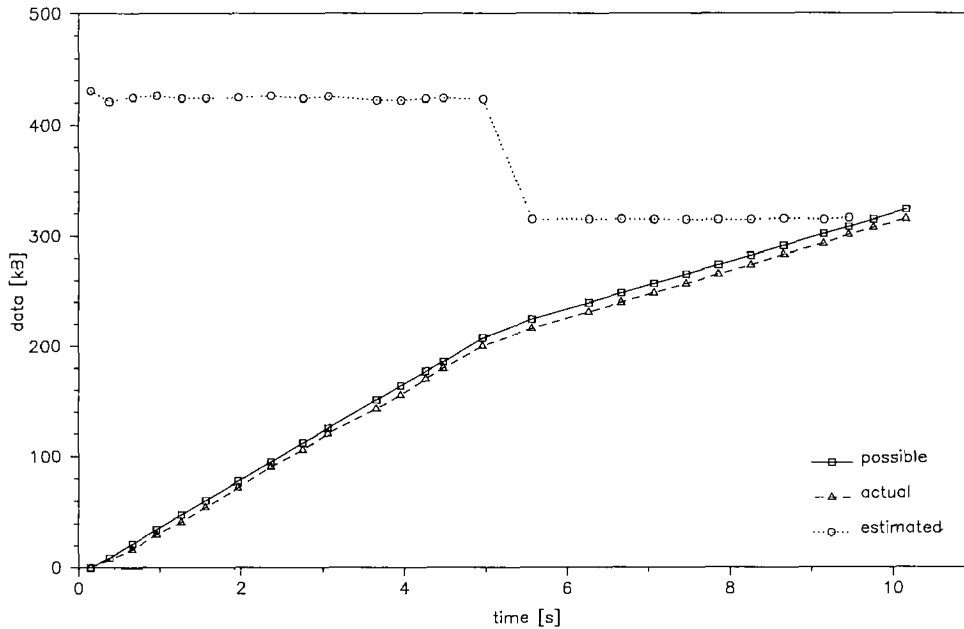


Figure 6.3: Data volume transmitted in *Step-Down* scenario.

curve named “possible”. The sharp drop in bandwidth at $t = 5$ s is absorbed almost without loss of transmission possibilities. Loss of transmission possibilities, which is characterized by the vertical difference between the curve showing the data theoretically transmittable (“possible”) and the data actually transmitted (“actual”), can be caused by prepare or control loop overhead. The difference at the end of the transfer reflects the overall utilization of the available bandwidth. The bandwidth utilization is 97.4% (see also Table 6.1).

The curve depicting the control loop's estimate of the total amount of data transmittable within the time limit (“estimated”) shows that the adaptation at $t = 5$ s takes place fairly swiftly (within half a second). The estimate is based on the amount of data already transmitted, the monitor's estimate of the available bandwidth $bw(t)$ and t_{left}. Even though new bandwidth estimates are available each 0.1 s, the adaptation takes place only after ≈ 0.5 s. With the simple adaptation policy employed (Section 6.1.3), *agility* is limited by the number and the spacing of adaptation points, that is, in the case of Chariot the number of images to transmit and the time required to transmit a single image ($t_{trans}(img_i)$). As shown in Section 4.8 more sophisticated adaptation policies, e.g., using progressive delivery, can help improve the agility of the network-aware application[4].

Figure 6.4 plots the control loop's error variable t_{diff} that drives Chariot's adaptation. The two horizontal lines at t_{diff} = 0.2 s and $t_{diff} = -0.2$ s represent the tolerance interval specified. The “time difference” plot shows that in fact two different adaptation events occurred

[4]There are two issues to note in Figure 6.3 (and similar figures that follow). First, the curves start at $t \approx 0.15$ seconds. The gap between $t = 0$ when the request is received by the Chariot server and $t \approx 0.15$ when the server enters the control loop, reflects the time used to parse the request message and locate the images to be delivered. Because non-adaptive response delivery witnesses the same overhead, we deliberately start counting the data that is transmittable (curve “possible”) after this period. This choice allows us to more accurately assess the performance of adaptation in terms of bandwidth utilization. Second, the markers indicate the progress of response delivery. E.g., triangles (curve “actual”) denote when each of the 25 images has been delivered. Circles (curve “estimated”) reflect adaptation points.

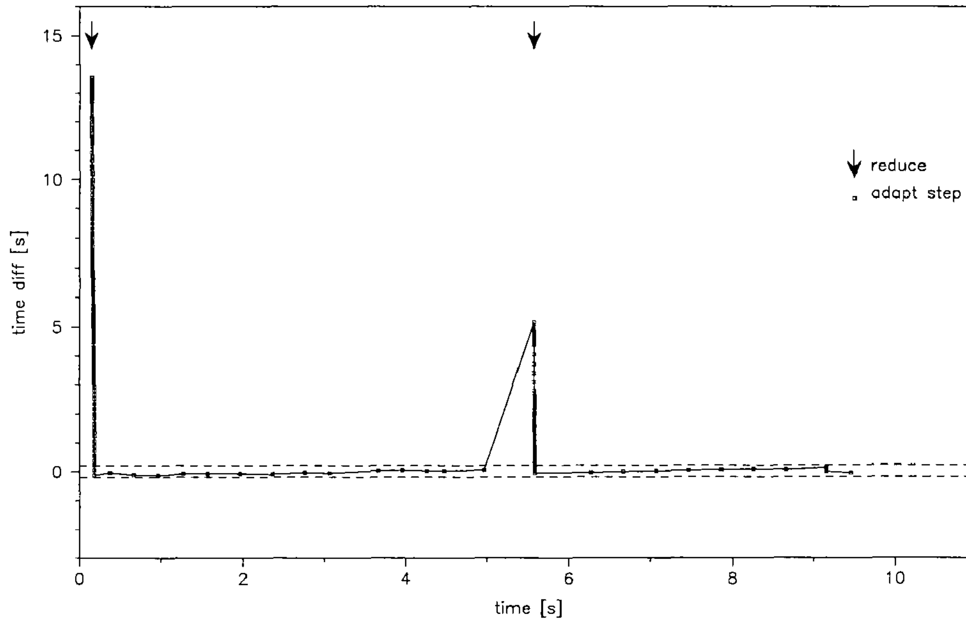


Figure 6.4: Time difference (t_{diff}) plot for *Step-Down* scenario.

Experiment	Time limit deviation [%]				Bandwidth utilization [%]			
	median	SIQR	min	max	median	SIQR	min	max
Step-Down	1.7	0.0	1.7	2.6	97.4	0.0	95.9	97.5
Internet	2.9	2.0	−1.3	6.9	78.1	5.6	74.7	91.9

Table 6.1: Summary statistics for the 50 runs of the step-down and the Internet experiment.

(adaptation is necessary when $|t_{diff}| > 0.2$ s). Adaptation events are emphasized by arrows (arrow down reflects reduction; arrow up would reflect expansion). First, around $t = 0$ s initial adaptation steps are necessary to reduce the 1.03 MB to the 430.8 kB estimated to be transferable. Second, due to the sharp bandwidth drop at $t = 5$ s, t_{needed} and hence t_{diff} increase by approximately 5 seconds; this drop is compensated in subsequent reduction steps. Note, adaptation steps might also be necessary if the server load changes or if the estimates for resource consumption are inaccurate, however, we specifically tried to minimize such effects (see Section 6.2.1). Section 6.7 discusses the effects of inaccurate estimators.

Table 6.1 shows that the variance in both performance metrics is minimal for the 50 experiments conducted—the semi-interquartile range (SIQR) is 0 for both metrics.

Internet traffic

Figure 6.5 shows that Chariot is even capable of dealing with frequent oscillations in the available bandwidth as present on today's wide-area network paths. The time limit is exceeded by 2.9% (just slightly more than tolerated). The results in Table 6.1 indicate that there are runs of the Internet experiment where the time limit is respected (column "min"). Note, however, that the penalty in terms of transmission possibilities lost is higher than in the previous case (utilization = 78.9%). The curve depicting the data volume transmittable ("possible") relates to the bandwidth waveform shown in Figure 6.2(b). Careful examination of the curve plot-

ting the data effectively transmitted reveals two cases (at $t \approx 0.5$ s and after $t \approx 8$ s) where transmission lulls had to be accepted. The reason is that in these cases P_{trans} for image i finished before the concurrently executed phase P_{prep} for image $i+1$ and thus had to wait before starting transmission of image $i+1$. The causes for this behavior can be twofold: Either $c_{prep}(img_{i+1}) > t_{trans}(img_i)$, in which case a more sophisticated adaptation policy could try to reorder the images in the request list to avoid communication idle time, or the server load is too high, such that $t_{prep}(img_{i+1}) = load \cdot c_{prep}(img_{i+1}) > t_{trans}(img_i)$. Since we run the experiments in an unloaded system (i.e. $load = 1$), the second cause does not apply here.

Figure 6.6 shows the time difference plot for the Internet experiment. It indicates that the adaptation events (depicted by arrows) closely reflect the fluctuations in bandwidth (Figure 6.2(b). Moreover, it helps explain the transmission lulls observed: At $t \approx 0.5$ s, the initial (small) reduction from 1.03 MB to the estimated 919.4 kB transferrable must be distributed evenly among all the 25 images, because the images are assumed to be equally important. As a consequence, every image must be reduced (by a small amount), which means that no image can be transmitted right away and $t_{prep}(img_1)$ has to be awaited before sending the first byte. This observation calls for more sophisticated adaptation policies that aim at reducing the risk of encountering transmission lulls, in particular during the start-up phase (see Section 4.7).

The performance loss in terms of bandwidth utilization experienced around $t \approx 8$ seconds is less easily explained. The numerous adapt events after this point and the fact that t_{diff} stays above the tolerance indicates that there is no adaptation potential left that could be exploited. In other words, all the remaining images have been forced to minimal quality beforehand. As a consequence, the images to be transmitted are small and the transformations to be applied are relatively costly, which leads to the transmission lulls observed. The lack of adaptation potential is also the reason why the time limit is exceeded in this case. There are various reasons that can cause such a situation. The most prominent cause in this example is that the bandwidth is over-estimated at $t \approx 4$ s, which results in a overly optimistic adaptation decision and a transmit phase that lasts (3 times) longer than predicted (up to $t = 6$ s). After that point there is only one alternative to keep the damage (in terms of exceeding the time limit) small: to reduce the remaining images to minimal quality.

The presumption that adaptation performance can be quite sensitive to the accuracy of the bandwidth estimation is reinforced by the following observations. Table 6.1 shows that there is a non-negligible variance in the performance of the 50 Internet experiments conducted. While the time limit is never exceeded by more than 7%, there is a much higher variance in terms of bandwidth utilization. Comparing the "median outcome" (experiment A) of the Internet experiment as presented above (Figures 6.5 and 6.6) with the "best outcome" (experiment B, Figures 6.7 and 6.8) indicates that the adaptation process can be quite sensitive to timing aspects—at least with the simple bandwidth estimator used here. For some reason, not relevant to the discussion here, the last expand decision before $t = 4$ s is made 0.1 s earlier in experiment B than in experiment A. At that time, the bandwidth reported was slightly lower and better reflecting the ensuing low bandwidth period (see Figure 6.2(b)). As a result, the adaptation decision made in experiment B was more conservative[5], the transmission phase did not last that much longer

[5]The last expand decision before $t = 4$ s is the second last adaptation decision before the zero-bandwidth period around $t \approx 5$ s. Since P_{prep} and P_{trans} operate in a pipelined fashion, this second last adaptation event before the zero-bandwidth period is important as it decides on the final size of the image to be transferred during that period.

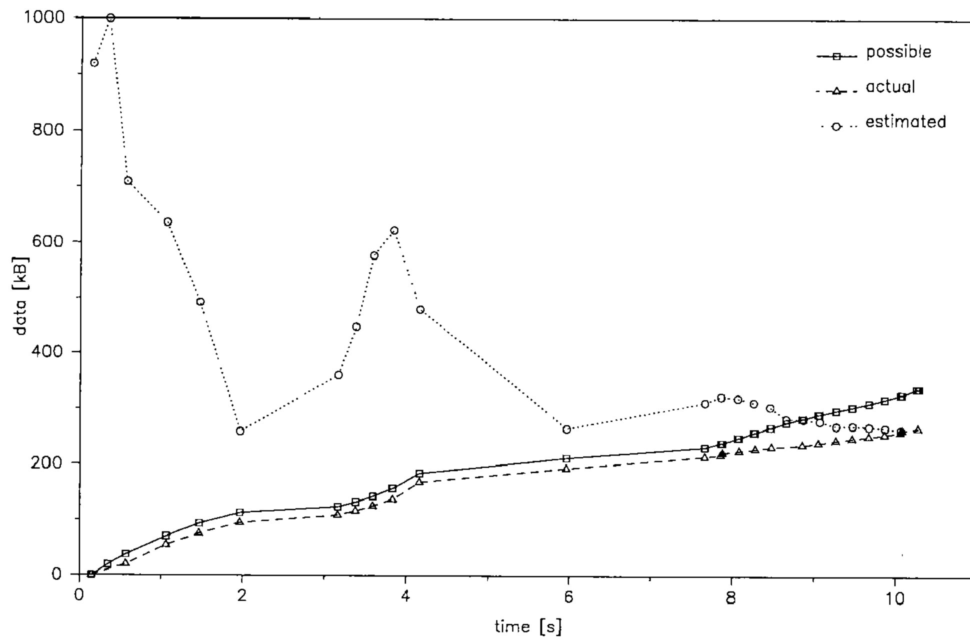


Figure 6.5: Data volume transmitted from Zürich to Linz (median utilization).

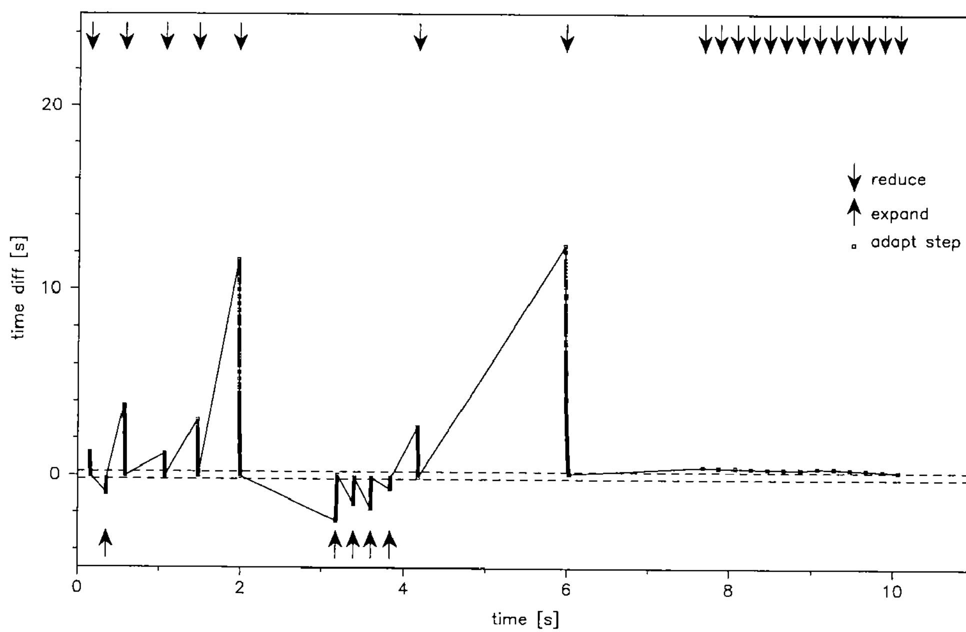


Figure 6.6: Time difference (t_{diff}) plot for Internet example (median utilization).

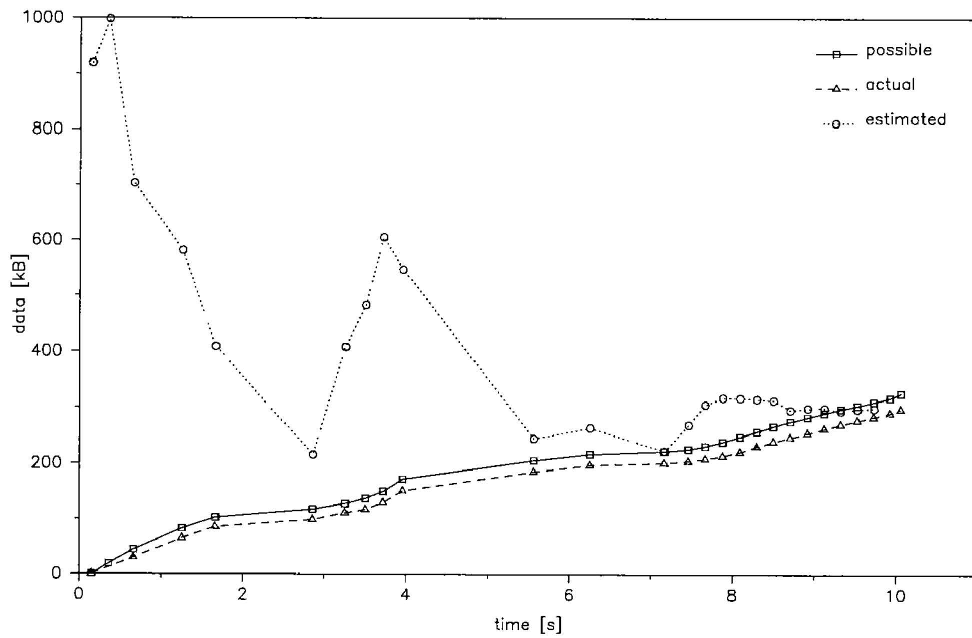


Figure 6.7: Data volume transmitted from Zürich to Linz (maximal utilization).

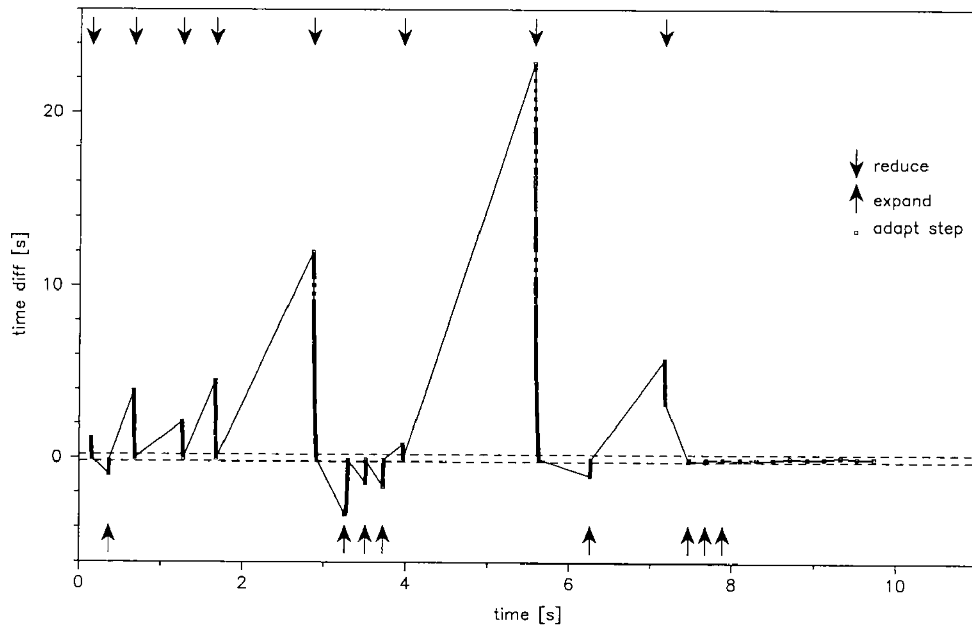


Figure 6.8: Time difference (t_{diff}) plot for Internet example (maximal utilization).

than predicted, and consequently there was enough adaptation potential left to deal gracefully with the bandwidth fluctuations that ocur after $t \approx 8$ s.

Summary

To wrap up, we can state that the model-based adaptation proposed in this dissertation is able to fulfill its goals, i.e., that adaptation is able to meet a user-imposed time limit while achieving high bandwidth utilization—a prerequisite for achieving high quality responses. However, the illustrative examples studied also reveal problems to be addressed by a more in-depth evaluation:

Adaptation can be quite sensitive to timing aspects and in particular to the accuracy of bandwidth estimation. Significant differences in performance and in performance variability between the two scenarios can be observed. The adaptation policy needs to be revisited to reduce the risk of transmission lulls.

Furthermore, we note that application agility needs to be improved to react more swiftly to changes in resource availability. Since agility is not effected by the framework, but mereley depends on the concrete application, and since techniques to improve agility (e.g., progressive data encoding) have been studied by other researchers (e.g., [66]) and discussed in Section 4.8, we refrain from elaborating on the issues of agility any further[6].

From these two experiments, it is unclear in which situations adaptation works well and under what circumstances it performs less favourably or in what situations it may even fail to provide the service requested, even though the boundary conditions on quality of the objects and the time limit were sufficiently coarse to allow for a timely adaptation. Network conditions are provably very diverse in today's Internet. So are most likely the requests that a network-aware server (e.g., a Chariot server) may have to process in a real-world environment. Therefore, we need to gain more insight into the dynamics of the adataption algorithm and develop a more profound understanding of the factors that influence the performance of adaptive systems.

In Section 6.3 we first assess the effect of framework-external factors as sketched in Section 6.1.2 on the performance of our system and in particular we focus on the CPU-bandwidth tradeoff to be made when adapting (Section 6.4). In Section 6.7 we turn to evaluating the effects of framework-interal factors, such as the accuracy of the resource estimators and the adaptation policy, on performance.

6.3 When does adaptation work?

This section identifies the key framework-external factors that influence adaptation behavior and quantifies their effect on performance.

[6]Clearly, agility can also be limited by the rate at which new bandwidth estimates become available. However, since aspects concerning bandwidth estimation are discussed in Chapters 7–9, the experiment setup has been chosen to ensure that the rate at which bandwidth estimates become available is not a limiting factor for adaptation performance.

6.3.1 Experiment

Section 6.1.2 introduced the set of framework-external factors that are assumed to influence the effectiveness of a network-aware system. Since we cannot simply assume that the factors are independent, that is, that the effect of their interaction is negligible, we first must identify primary factors, their interactions and their effects on the response variables, i.e., the performance metrics defined in Section 6.1.1. We can then proceed to study performance as a function of a single (primary) factor for instance. We use a *full factorial design* for the experiments [82]. The main problem of a full factorial design in our case is that there are too many factors and factor levels—most factors can be varied continously over ranges covering several orders of magnitude—to explore all of them. To simplify the search for key factors we restrict the experiment to a $2^k r$ full factorial design, where k is the number of factors, and r is the number of repetitions for each of the 2^k experiments. This means that the number of factor levels for each of the k factors must be reduced to 2, e.g., "high" vs. "low" bandwidth. The problem with such a simplification is that it makes sense only if it can be assumed that the effect of a factor is *unidirectional*, that is, the performance either continuously increases or continuously decreases as the factor is increased from the low level to the high level. Based on the discussion of each of the factors in Section 6.1.2, we expect this to be the case.

6.3.2 Factor levels

Since our primary focus is to evaluate the framework's network-awareness, we exclude the factor host load and conduct our experiments in an unloaded system. The levels for the remaining factors are chosen as follows (see Table 6.5 for a summary).

Network

The levels of the factor *bandwidth* differ by an order of magnitude. *Low bandwidth* reflects approximately ISDN speed and *high bandwidth* reflects approximately the speed of ADSL (asymetric digital subscriber lines). The term "approximately" is appropriate because we want to have the same average bandwidth for both levels of bandwidth *volatility*. The two levels of *volatility* used comprise *zero volatility* (or constant bandwidth) and a repeated series of impulses (*high volatility*), which means that the bandwidth repeatedly toggles between a constant low level and a constant high level. The bandwidth stays at each level for one second before changing to the other level. For the low bandwidth case the bandwidth curve toggles between 64 kbit/s and 128 kbit/s. Such a choice may seem fairly artificial, however, consider that an ISDN link has two 64 kbit/s channels and that both channels can be used in parallel, if there is no competing traffic allocating one of the channels. Thus, it is plausible that the bandwidth available to an application toggles between the two levels. Since the average bandwidth for the non-volatile and the volatile case should be identical, we have 96 kbit/s for the constant low bandwidth case. For the high bandwidth case, the bandwidth toggles between 0.64 Mbit/s (ADSL) and 1.28 Mbit/s in the volatile scenario, and therefore a constant bandwidth of 0.96 Mbit/s is used in the non-volatile scenario.

Request		Low adaptation potential					High adaptation potential				
		Image size distribution				Tot. size	Image size distribution				Tot. size
ID	n	min	max	μ	σ	d_{max}	min	max	μ	σ	d_{max}
C_{12}	12	8.6	11.3	10.2	1.0	123	45.1	54.7	50.8	3.6	609
V_{12}	12	5.7	98.5	19.3	25.1	232	33.1	242.7	70.3	55.1	844
C_{100}	100	7.5	20.0	13.5	3.2	1352	35.5	81.8	60.5	10.5	6048
V_{100}	100	5.2	336.8	43.2	55.4	4320	11.8	965.5	92.3	125.9	9228

Table 6.2: Request properties. All size information in kB.

End-system

To study the effect of *CPU bandwidth*, we run the network-aware server on two different platforms, on a moderatly fast 200 MHz Pentium Pro workstation with 64 MB RAM running NetBSD 1.3 and on a faster dual processor 300 MHz SPARC Ultra 4 server with 1GB RAM running SunOS 5.6. To address the question whether network bandwidth is better traded off with CPU or with storage, we introduce a third CPU level by means of an "infinitely fast server", that is, a server that precomputes all the versions of all the objects in the repository and thus can provide any version at an infinitesimally small (i.e. zero) cost. Since we cannot have three levels for a factor in a 2^k experimental design, we introduce a new factor termed *"prepare required"* with the two levels *yes* and *no*.

Application

Again, we use Chariot as our vehicle for experimentation. For the sake of simplicity we deal only with JPEG images in the requests and register only JPEG compression to be used as transformation algorithms. (We do not expect the choice of transformation algorithms to play an important role except for the cost of transformations and the reductions in size and quality they can achieve. The effect of different costs is discussed in Sections 6.4 and 6.5.) The boundary condition on the minimal quality of the images to be delivered is defined by a fixed JPEG quality factor of 10. The tolerance on the time limit is set to $\pm 2\%$ (of t_{left}).

Request. The requests used in this chapter were obtained by using query images that have been used for the evaluation of the color and texture features integrated in Chariot [40]. For each of the queries the search engine computed request lists of varying size (12, 25, 50 and 100 images). For each image in a request the search engine provided a similarity measure reflecting the relevance of the image with respect to the query image. This setup enables us to discuss aspects of utility in Section 6.6. The levels for the factor *request size* are 12 images for a *small* request and 100 images for a *large* request. The factor *object size distribution* takes on the two levels: *constant* and *variable*. Table 6.2 summarizes properties of the requested images for each of the four requests for two levels of *adaptation potential* as explained below. The columns min, max, μ and σ report the size of the smallest and largest image, as well as the average size of the images in the request and the standard deviation of the image sizes. The ratio of σ and μ partitions the requests into the *constant* sized requests C_{12} and C_{100} (with a ratio $\ll 1$) and the

variable sized requests V_{12} and V_{100} (with a ratio ≈ 1 or > 1). The indices of the request IDs indicate the number of images in a request.

Adaptation potential. To study the effect of various levels of *adaptation potential*, all the objects of the four requests are available in two versions. First, encoded with a JPEG quality factor of 100 and second, stored in their default JPEG quality factor (average JPEG quality factor 75). The first choice provides maximal quality images while the second choice makes a (reasonable) quality-size tradeoff. Since the boundary condition on the minimal quality acceptable (by the user) is set to a fixed JPEG quality factor (10) for our experiments, the quality 100 images represent *high* and the quality 75 images represent *low adpatation potential.* The columns d_{max} in Table 6.2 reflect the total size of the images requested in kB for the two levels of adaptation potential chosen. The boundary condition on the minimal image quality implicitely determines the minimal amount of data that must be transferred to deliver a request (columns d_{min} in Tables 6.3 and 6.4) and therefore also determines the adaptation potential ($\alpha_{pot} = d_{max}/d_{min}$, see columns α_{pot}). For the requests used in this experiment the two levels of adaptation potential differ by a factor of $\approx$ 2–4.

Adaptation granularity. Based on the observations on size reductions achievable with JPEG compression, which were made when constructing size models for the transformations used in our experiments (Section 5.1.3), we identified the following two levels of granularity. For the level with *coarse adaptation granularity* the JPEG quality factors 95, 90, 85, 80, 50, and 10 are used. The quality factors applicable in the *fine adaptation granularity* scenario are 99, 98, 97, 96, 95, 90, 85, 80, 70, 60, 50, 40, 30, 25, 20, 15, and 10. The extremely fine granularity for high values of the JPEG quality factor is chosen because the reduction in size is quite considerable at these levels (see Figure 6.15). The comparatively fine granularity for JPEG factors ≤ 30 has been chosen based on observations made by McIlhagga et al. [116], who claim that at these levels the "image quality" deteriorates considerably faster with decreasing JPEG quality than for higher JPEG quality factors. (The aspect of quality is discussed in Section 6.6.) For these two choices of adaptation granularity each image of the n images in a request can be delivered in at most 7 or 18 different versions respectively (including the "original" version). The columns μ_v in the Tables 6.3 and 6.4 show the average number of versions per image for each of the requests, adaptation potentials and adaptation granularities. The column "# comb." approximates the size of the adaptation space, reflecting the (vast) number of possible combinations of image versions for each request ($\approx {\mu_v}^n$). Note that μ_v both depends on the adaptation granularity *and* the adaptation potential.

Adaptation required. To find appropriate levels for the factor *adaptation required*[7] that reflect how hard the control loop has to work to meet the time limit, we not only need to consider the ratio of d_{trans} and d_{max} (= $d_{left}(0)$), but must take properties of the adaptation space into account (as discussed in Section 6.1.2). To illustrate the concern raised in Section 6.1.2 we compute the histograms for the response size distribution for all possible responses for a particular request, adaptation potential and adaptation granularity. The response size reflects the total size of the images returned in response to a particular user request. The histograms in

[7] We must pick two levels for α (α_{medium} and α_{high}), where $d_{trans} = d_{min} + (d_{max} - d_{min}) \cdot \alpha/100$.

Request		Transformations				Coarse granularity				Fine granularity			
ID	d_{max}	d_{40}	d_{10}	d_{min}	α_{pot}	# comb.	μ_v	n_{40}	n_{10}	# comb.	μ_v	n_{40}	n_{10}
C_{12}	123	75	52	44	2.8	5.3e+05	3.0	10.3	0.1	2.8e+11	9.0	14.7	0.0
V_{12}	232	134	85	69	3.4	7.1e+05	3.1	2.5	0.2	3.1e+11	9.1	6.3	0.2
C_{100}	1352	813	544	454	3.0	5.2e+47	3.0	0.0	0.0	2.7e+95	9.0	2.5	0.0
V_{100}	4320	2264	1236	893	4.8	1.3e+61	4.4	9.6	0.0	2.1e+104	11.8	17.2	0.0

Table 6.3: Transformation properties for requests with low adaptation potential. Size information in kB.

Request		Transformations				Coarse granularity				Fine granularity			
ID	d_{max}	d_{40}	d_{10}	d_{min}	α_{pot}	# comb.	μ_v	n_{40}	n_{10}	# comb.	μ_v	n_{40}	n_{10}
C_{12}	609	270	100	43	14.0	1.4e+10	7.0	4.9	0.0	1.2e+15	18.0	3.8	0.2
V_{12}	844	378	145	68	12.5	1.4e+10	7.0	5.0	0.1	1.2e+15	18.0	4.5	0.3
C_{100}	6048	2688	1008	447	13.5	3.2e+84	7.0	8.5	0.0	3.4e+125	18.0	2.2	0.0
V_{100}	9228	4226	1726	892	10.3	3.2e+84	7.0	15.0	0.0	3.4e+125	18.0	10.1	0.0

Table 6.4: Transformation properties for requests with high adaptation potential. Size information in kB.

Figure 6.9, exemplarily depict the normalized[8] response size distributions for requests C_{12} and V_{100} (low adaptation potential, two levels of adaptation granularity). The dashed lines reflect the maximal and the minimal possible response size ($d_{min} \leq d_{trans} \leq d_{max}$). The figure indicates that there can be quite significant differences in the respsonse size distributions between the different requests. Constant-sized requests tend to have more symmetric distributions than variable-sized requests, and fine adaptation granularities seem to produce smoother distributions. Most importantly, however, the histograms show that there are in fact ratios of adaptation required that are less easily achievable than others because there are substantially fewer possible responses that could fulfill the particular time limit.

In principle, we would have to disentangle the factors "reduction ratio" and "likelihood to achieve the ratio". Because the factors are not independent, we must choose the two levels for the factor *adaptation required* so as not to violate the unidirectionality assumption of the underlying 2^k design. This means that the *high* level of adaptation required must achieve a high reduction ratio (α_{high}) *and* there must only be a small set of responses applicable. On the other hand, the *medium* level should only require a moderate amount of reduction ($\alpha_{high} < \alpha_{medium}$) and there should be a considerably larger set of responses that could fulfill the request. The analysis of the histograms for all the requests, adaptation potentials, and adaptation granularities considered in this experiment leads us to pick $\alpha_{high} = 10$ and $\alpha_{medium} = 40$. Tables 6.3 and 6.4 list the response sizes corresponding to the choice of α_{high} (d_{10}) and α_{medium} (d_{40}). The columns n_{10} and n_{40} report which percentage of the possible responses lie within a factor $1 \pm \varepsilon$ of d_{10} and d_{40} respectively, where ε is the tolerance on the time limit (0.02 in our experiments). Note that a value of 0 in one of these columns does not necessarily mean that the set of possible responses is empty, but that only $\ll 0.1\%$ of all the responses can fulfill the particular request.

Table 6.5 summarizes all the factors and factor levels used in the $2^k r$ experiment ($k = 9$). The factors *RS* and *RD* are client-specific and determine which images are requested from the

[8] Normalized means that the area below the histogram equals 100 (%).

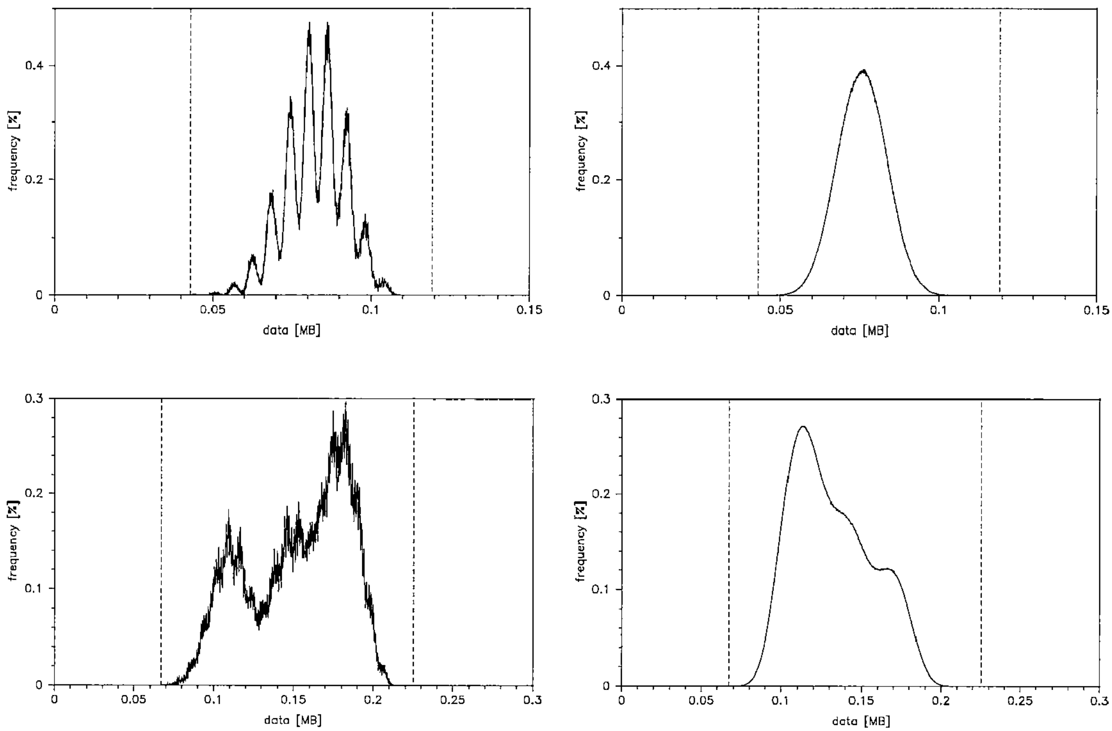


Figure 6.9: Normalized histograms of response sizes for requests C_{12} (upper figures) and V_{12} (lower figures) for the two levels of adaptation granularity *coarse* (figures on the left) and *fine* (figures on the right). Both requests reflect the scenario with low adaptation potential.

Factor	Abbrev.	Level -1	Level 1
Bandwidth	BW	low (ISDN)	high (T1)
Volatility	BV	none	high
Prepare required	PREP	yes	no
CPU	CPU	i686/NetBSD	2 uSPARC/SunOS
Request size	RS	small (12)	large (100)
Size distribution	RD	constant	variable
Adaptation potential	AP	low (∅ quality 75)	high (∅ quality 100)
Adaptation granularity	AG	coarse (≤ 7)	fine (≤ 18)
Adaptation required	AR	medium (40%)	high (10%)

Table 6.5: Factors and factor levels for the $2^k r$ experimental design ($k = 9, r = 5$).

server. The factor *CPU* indicates which server to contact. The factors *AP*, *AG* and *PREP* are server-specific in that they determine which version of the images requested must be delivered (factor *AP*), what transformation algorithms are applied (*AG*) and whether all the possible versions of the images must be precomputed or not (*PREP*). The factors *BW* and *BV* are used to select the appropriate bandwidth replay trace to emulate the bandwidth available. The factor *AR* implicitly determines the time limit. The time limit for each of the 2^k experiments is computed as a function of the factors *RS*, *RD*, *AP*, *BW*, *BV* and *AR*. *RS*, *RD* and *AP* determine d_{max}, *AR* determines d_{trans}[9], and *BW* and *BV* determine how long it takes to transfer the d_{trans} bytes across the network. The tolerance on the time limit ε used in the experiments is defined as $max(0.02 \cdot t_{left}, 0.1)$ seconds, that is, the tolerance interval narrows as the transfer progresses. Each of the 2^k experiments is run 5 times (i.e., $r = 5$). The adaptation policy used is the same as in the examples in Section 6.2 and as described in Section 6.1.3.

6.3.3 Results

The evaluation methodology chosen (a 2^k factorial design with replications) allows us to determine the *effects* of the 2^k factors and their interactions on the performance metrics of interest according to the procedures described in [82]. The two metrics of interest in this section are the deviation from the time limit at the end of the image transfer and the utilization of the bandwidth available. The term t_{diff} is (re)used to reflect the relative deviation from the time limit ($t_{diff} = 100 \cdot (t_{used} - t_{allotted})/t_{allotted}$). The *importance* of a factor is measured by the proportion of the total variation in the response, i.e. in the performance metric of interest, that is explained by the factor. Since the experiments are repeated $r = 5$ times, the percentage of the total variation in the response that is due to experimental errors can be isolated. The variance of the sample estimates σ_e^2 can be used to compute confidence intervals for the effects of each of the factors and factor combinations (assuming that the errors are normally distributed with zero mean and variance σ_e^2). All the confidence intervals reported in the following are computed at a confidence level of 95%. A factor whose confidence interval for the effect includes 0 is statistically not significant.

How to read the results?

The following paragraphs explain how to read the results of the 2^k experiment. The results are presented in tabular form such as shown in Table 6.6. (This introduction closely follows the methodology described by Jain [82]). The interpretation of the results and conclusions from the results are deferred to separate sections below.

Table 6.6 lists the key factors effecting the t_{diff} metric in decreasing order of importance, that is, in decreasing order of the variation explained by the factor. Factors and factor combinations are separated and only listed if they explain more than 2% of the total variation. The row *mean* reports the mean t_{diff} for all the $2^k r = 2560$ experiments conducted and the 95% confidence interval for the mean. On average, the time limit is exceeded by 21.9%. The key factors that effect how well (or how badly) the control loop is able to meet the time limit are *bandwidth* (*BW*) and whether or not the images *need to be transformed* at run-time (*PREP*). Both factors

[9] Recall, that d_{min} is implicitly defined by the choice of the minimal quality acceptable.

explain around 14% of the variation (*BW* explains 13.9% and *PREP* explains 13.6% of the total variation). The effects are interpreted as follows: The value for the effect of a factor, e.g., the 18.7% for the factor *BW*, indicates that the factor effects the mean t_{diff} by $\pm$ 18.7%. For the level 1 of the factor *BW*, i.e. in case of a high bandwidth (ADSL) connection, the mean t_{diff} increases by 18.7% (to 40.6%). For the level -1 of the factor *BW*, i.e. in case of a low bandwidth (ISDN) connection, the mean t_{diff} decreases by 18.7% to 3.2%. A negative value for the effect, such as the -18.6% for the factor *PREP*, indicates that the mean is affected by $\mp$ 18.6%. For the level -1 of the factor *PREP*, i.e. in case transformations must be applied at run-time, the mean t_{diff} increases by 18.6%. On the other hand, in case all the versions of the images requested are precomputed, i.e. for level 1 of *PREP*, the mean t_{diff} is reduced to 3.3%.

Because the factors are not independent, we cannot conclude that in a situation with low bandwidth and precomputed images the mean t_{diff} would be -15.4% (i.e. $18.7 + 18.6 = 37.3\%$ lower than the average 21.9%). To draw conclusions about situations where a subset of the k factors are kept fixed (e.g., where *BW* and *PREP* are kept fixed), the effects of *all* the factor combinations that include *any* of the fixed factors (e.g., that include *BW* or *PREP*) have to be taken into account. For the interaction of *BW* and *PREP*, for instance, we see that level -1, which comprises the two combinations $(BW(-1), PREP(1))$ and $(BW(1), PREP(-1))$, results in a 18.8% higher mean t_{diff}, whereas level 1, which comprises the other two combinations, results in a 18.8% lower mean t_{diff}. $BW(-1)$ denotes level -1 of factor *BW*. Thus, for a particular situation, e.g., $(BW(-1), PREP(1))$ reflecting a scenario with low bandwidth and precomputed images, we would have a t_{diff} of $21.9(mean) - 18.7(BW) - 18.6(PREP) + 18.8(BW, PREP) = 3.2\%$ (not counting other factor combinations that include *BW* and *PREP*).

Since the analysis of the results for specific combinations of (fixed) factors is complicated considerably by such interaction effects, we will study the scenarios of interest in isolation and present the effects for the "reduced" factorial designs separately (instead of deriving them from the overall results).

Overall results

Time limit. Table 6.6 shows the key factors effecting the control loop's ability to meet the time limit. As mentioned above, the most important factors are the *bandwidth available* for the transfer and whether transformations are required at run-time (*prepare required*) or whether all the image versions have been precomputed. Less important, but still significant are the factors *adaptation required*, *CPU*, and *adaptation potential*. To qualify the importance of the key factors identified we must compare them to the percentage of variation explained by experimental errors. Experimental errors explain merely 1.1% of the total variation and are thus negligible. The effects of the key factors are fairly intuitive. The higher the bandwidth is, the less time there is to adapt and to reduce the quality of the objects and — since the degree of *adaptation required* is fixed and hence the transformation costs are constant — the more likely it is that the adaptation and transformation costs exceed the time limit. However, if the reductions in quality come for free, that is, if the CPU is extremely fast or if all the versions are precomputed, the penalty in terms of deviation from the time limit is marginal (3.3%; see Table 6.8). The effect of the factor *CPU* (-8.5% for the faster CPU) also supports this argumentation. Similarly intuitive is that the higher the *adaptation required* is, the higher are the transformation costs

Factor	Effect	Variation	Conf. Interval
Mean	21.9		(21.6, 22.1)
BW	18.7	13.9	(18.5, 18.9)
PREP	−18.6	13.6	(−18.8,−18.3)
AR	12.9	6.6	(12.6, 13.1)
CPU	−8.5	2.9	(−8.7, −8.3)
AP	−7.4	2.2	(−7.6, −7.1)
BW PREP	−18.8	14.0	(−19.0,−18.6)
BW PREP AR	−11.5	5.2	(−11.7,−11.2)
PREP AR	−11.3	5.1	(−11.6,−11.1)
BW AR	11.1	4.9	(10.9, 11.4)
BW CPU	−8.6	2.9	(−8.8, −8.3)
BW PREP CPU	7.9	2.4	(7.6, 8.1)
PREP CPU	7.8	2.4	(7.5, 8.0)
Errors		1.1	($\sigma_e = 5.9$)

Table 6.6: Key factors and their interactions (listed in decreasing order of "importance") and their effects on the t_{diff} metric. The upper part of the table lists the effects of individual factors; the lower part lists the effects of factor combinations. t_{diff} reports the relative deviation from the time limit (in %), i.e. $100 \cdot (t_{used} - t_{allotted})/t_{allotted}$. The "importance" of a factor is reflected by the percentage of variation in the t_{diff} measurements explained by the factor. Factors and interactions of factors are only listed if they explain more than 2% of the variation. The last column lists the confidence intervals for the effects at a confidence level of 95%.

Factor	Effect	Variation	Conf. Interval
Mean	87.9		(87.8, 88.0)
BW	−10.9	24.4	(−11.0,−10.8)
PREP	10.5	22.8	(10.4, 10.7)
CPU	3.3	2.2	(3.2, 3.4)
BW PREP	9.9	20.3	(9.8, 10.1)
Errors		1.5	($\sigma_e = 3.0$)

Table 6.7: Key factors and their interactions (in decreasing order of importance) and their effects on *bandwidth utilization* (in %): $100 - (possible - actual)/possible$. Factors listed if variation explained $\geq 2\%$.

bandwidth	t_{diff} [%]		utilization [%]	
	low	high	low	high
prepare required	2.9	77.9	98.2	56.5
no prepare required	3.4	3.2	99.4	97.5

Table 6.8: Summary results for $2^k r$ experiment classified according to the key factors *prepare required* and *bandwidth.*

and the worse is the performance "timewise". The factor *AR* makes a difference on the mean t_{diff} of $\pm$ 12.9% (for the comparatively small difference between the two degrees of adaptation required $\alpha_{high} = 10\%$ and $\alpha_{medium} = 40\%$). Furthermore, the fact that t_{diff} increases with decreasing *adaptation potential* confirms the presumption made in Section 6.1.2, which stated that the chances that the system can react to changes in resource availability should increase with higher levels of adaptation potential.

The 21.9% by which the time limit is exceeded on average represent quite a considerable and disturbing performance penalty. A closer inspection of the results reported in Table 6.6, which takes the interaction of factors into account, reveals that the adaptation mechanisms perform very favorably both in low bandwidth scenarios and in case no transformations must be applied at run-time, as the time limit for these three cases is merely exceeded by $\leq$ 3.4% on average (see Table 6.8). Only in the case of *high bandwidth* and *prepare required* does the performance degrade substantially (t_{diff} is 77.9%). Reasons for this performance penalty and possible remedies are discussed below.

Bandwidth utilization. Table 6.7 lists the key factors effecting the control loop's ability to make efficient use of the bandwidth available. For this metric, the situation is much simpler as there are basically only two factors, bandwidth (*BW*) and end-system conditions (*CPU, PREP*), that effect performance. The mean bandwidth utilization for all the $2^k r$ experiments is 87.9%. The factors *BW* and *PREP* both account for about 23–24% of the total variation and effect bandwidth utilization by $\mp$10.9% and $\pm$10.5% respectively. The effects observed are analogous to the effects on the t_{diff} metric, that is, low bandwidth or fast CPU result in higher utilization. Experimental errors are again negligible as they merely account for 1.5% of the variation.

The non-optimal bandwidth utilization of 87.9% is again mainly a consequence of the two primary factors *PREP* and *BW*, as well as their interaction. As shown in Table 6.8 scenarios with low bandwidth or no transformation costs achieve a nearly optimal bandwidth utilization ($\geq$ 97.5%), whereas the high bandwidth situations that required prepare activity performed less favorably (utilization 56.5%, see discussion below).

Robustness. It is interesting to note that the key factors effecting t_{diff} and the utilization of the bandwidth available do *not* include factors such as request properties (request size and size distribution) or transformation properties (adaptation granularity). This result means that these factors do not seem to effect performance significantly. More astonishingly, whether bandwidth fluctuates or not does not make a notable difference in performance either. This finding is important because it indicates that model-based adaptation does not suffer instability problems regardless of how many of the factors influencing adaptation decisions vary.

The result is somewhat surprising, because the examples presented in Section 6.2 seem to indicate that high bandwidth volatility (such as present in the Internet scenario) is problematic for the effectiveness of adaptation. Care must be applied when drawing final conclusions here, as these results may be simply an artefact of the choice of the bandwidth traces used in the $2^k r$ experiment. Note for instance that real-world bandwidth traces can exhibit considerably higher volatility than the synthetic traces used here. Nevertheless, the result is promising, as it indicates that the framework is robust enough to deal with medium levels of bandwidth volatility without negative impact on adaptation performance. The aspect of volatility is investigated further in Section 6.7.

In summary, the results in terms of t_{diff} and in particular *bandwidth utilization* are encouraging, because they indicate that the adaptation mechanisms are fairly robust in delivering reasonably high quality responses regardless of most of the factors studied in this experiment. As a consequence, performance considerations boil down to the *communication-computation* tradeoff striven for by network-aware content delivery. Section 6.4 investigates this tradeoff in more detail.

Detailed results: high bandwidth, prepare required

Experiments with the factors *high bandwidth* and *prepare required* experience considerable performance problems, both in terms of deviations from the time limit and in terms of bandwidth utilization. The following paragraphs discuss causes and possible remedies.

As far as the metric t_{diff} is concerned, performance problems clearly arise when the costs for the entire adaptation process exceed the difference of time limit and the time needed to transmit a minimal quality response. Bandwidth utilization suffers mainly in case of prepare phase–transmit phase mismatches and can therefore be adversely affected even before the adaptation costs exceed the difference above. As will be detailed in Section 6.5, the costs for adaptation mainly comprise transformation costs and decision making overhead. There are other sources of overheads though, for example, the initial request processing or potential start-up latencies (see Section 4.7).

A key to understanding the results is the fact that we chose to use two (fixed) levels for the factor *adaptation required* (α) for the experimental design. A fixed α implies that the time limit decreases with increasing bandwidth. Thus, high bandwidth means short time limits. Moreover, a fixed α implies constant transformation costs (for varying bandwidth levels).

Thus, there are basically two factors that can adversely effect performance: (i) small time limits, and (ii) high bandwidths. First, the shorter the time frame allotted for response delivery, the more likely it is that the fixed costs (such as decision making overhead) cannot be amortized over the short time period and hence the more likely it becomes that the time limit is exceeded. Second, the higher the bandwidth, the more likely are prepare–transmit phase mismatches given constant transformation costs. (Such mismatches affect bandwidth utilization if $t_{prep} > t_{trans}$.) This reasoning is supported by the numbers on idle transmit times (t_{idle}), observed for the experiments, where $t_{idle} = \sum_{i=1}^{N} max(t_{prep}(obj_{i+1}) - t_{trans}(obj_i), 0)$. Considering only the experiments with *prepare required* we find that the low bandwidth experiments suffer only 1.4 seconds idle transmit time on average, whereas the mean t_{idle} in the high bandwidth experiments is 6.5 seconds. Since the average time limit for the high bandwidth experiments is

only 7.8 seconds, it is clear why the performance is lousy.

How could this unsatisfactory situation be remedied? An obvious approach is the reduction of costs. There are several options to reduce costs. First, optimistic start-up strategies (see Section 4.7) can reduce start-up latency. Second, performance tuning may allow to reduce decision making overhead and request processing latencies. A third option would concern transformation costs. How can the transformation costs be reduced (if the degree of adaptation required (α) is kept fixed and the transformations used are assumed to be optimized for execution time)? Changing the adaptation policy (Section 6.1.3) from evenly distributing the data and quality reductions among all the transmission objects, e.g., to concentrate the reductions on as few objects as possible (e.g., the least important ones) may be able to reduce transformation costs to some extent (by 16% in the case of the Chariot experiments). Noting that t_{idle} is effected significantly by the factor *request size distribution*—t_{idle} is significantly smaller for the requests with constant-sized objects—we expect that a more sophisticated adaptation policy that tries to rearrange the objects within a request would be able to minimize t_{idle} and hence would improve performance.

6.4 Communication-computation tradeoff

The previous sections identified bandwidth and CPU power as the dominant factors effecting performance. Furthermore, the results of the $2^k r$ experiment indicate that the other factors (request properties, transformation properties, etc.) often effect adaptation performance only marginally. This implies that the adaptation mechanisms are fairly robust in delivering reasonably high quality responses regardless of most of the factors studied in the experiment. As a consequence, performance considerations boil down to the *communication-computation* tradeoff striven for by network-aware content delivery. This section discusses this tradeoff.

6.4.1 Experiment

The results of the $2^k r$ experiment are somewhat counter-intuitive as performance dwindles with increasing bandwidth. After all, the idea behind the type of network-aware delivery proposed in this dissertation is to compensate low bandwidths with reductions in quality and size of the data to be delivered to meet a user-specified time limit. For a particular time limit we would expect that the smaller the bandwidth, the higher the "compression" ratios required and the higher the transformation costs. Thus, we would expect low bandwidths to be problematic as far as performance is concerned.

Therefore, instead of keeping the degree of adaptation required constant, as in the case of the $2^k r$ experiment, we ought to study scenarios where the time limit is kept constant and the bandwidth varies, because this setup more closely reflects the service model provided to the user (Section 3.2). (We did not do so in the $2^k r$ experiment because the range in which the bandwidth can vary is limited by the adaptation potential of the request. The adaptation potential can be fairly low as can be learned from Table 6.3 for example.)

To study the computation-communication tradeoff in more detail we conduct the following experiment. Using a fixed request (50 constant-sized JPEG images, high adaptation potential) and a fixed time limit of 30 seconds (to avoid problems that result from too small a time limit,

see Section 6.3.3) we study performance as a function of bandwidth and CPU power. With the request used, the bandwidth can be varied from 85 kbit/s up to 1.1 Mbit/s. If we chose bandwidths < 85 kbit/s, we could not help but missing the time limit because even a minimum size response (d_{min}) would require more than 30 seconds to be delivered. If we chose bandwidths > 1.1 Mbit/s, no adaptation would be required, the images could be sent unchanged and the transfer would end before the time limit expires. For the sake of simplicity, only constant bandwidth traces are used for this experiment.

How can the factor CPU power be varied? Using different platforms for the server is quite resource-intensive and may only provide a limited range of performance levels, which are hard to quantify since numerous factors effect performance of a computer system for a given non-trivial workload. Using different levels of background host loads is problematic because t_{prep} may not linearly depend on c_{prep} and $load$, but may also depend on other factors, such as the cumulative resource consumption of the background processes [119] or the specific operating system (and the scheduling discipline) used[10]. These considerations lead us to emulate the effect of a CPU n times slower than a reference CPU by using an n times more costly transformation algorithm. Even though the transformation algorithms are easily replaceable in our framework, it may prove difficult to find appropriate transformations that meet the requirements on CPU resource consumption. For our experiments, we solved the problem by replacing the actual transformations (e.g., JPEG compression) by an artificial transformation algorithm that would consume a specified amount of CPU resources and would reduce the transmission objects by a specified amount of bytes[11]. This approach is clearly limited, as it neglects many of the complexities of real workloads. On the other hand, the approach has the nice property that it allows to emulate the performance of (n times) *faster* servers by using (n times) less expensive transformations.

The results discussed in the following are obtained by running the Chariot server on a 200 MHz Pentium Pro PC with NetBSD 1.3. The y-scale of the figures presented below reflects CPU power. The y-scale is indexed and 1 indicates the effective transformation costs for JPEG compression. A value of y means $1/y$-times the effective costs for JPEG compression. $y = \infty$ indicates an infinitely fast CPU, that is, reports the results for a configuration where all the possible responses are precomputed. For each bandwidth level and each level of CPU power the experiment has been repeated 5 times. The figures report the median performance (variance due to experimental errors is very low).

6.4.2 Results: Constant time limit

Time limit. For a constant time limit and a fixed request we would expect the smaller the bandwidth gets, the higher the transformation costs are and thus the more likely it is that the time limit is exceeded. The time limit is exceeded if the transformation costs are larger than the difference of the time limit and the time needed to transmit a minimum size response. Further-

[10] Section 4.4 suggests a simple linear relation between $load$ and c_{prep}, however, mainly for the sake of simplifying the discussion. Future work must show how the duration (wallclock time) of a task can be estimated reliably. Accurate predictions of host load may provide a basis here [42].

[11] The application-specific resource models for CPU consumption of (c_{prep}) and the size reduction achieved by a particular transformation provide the necessary information to emulate the resource requirements of an actual transformation, e.g., JPEG compression.

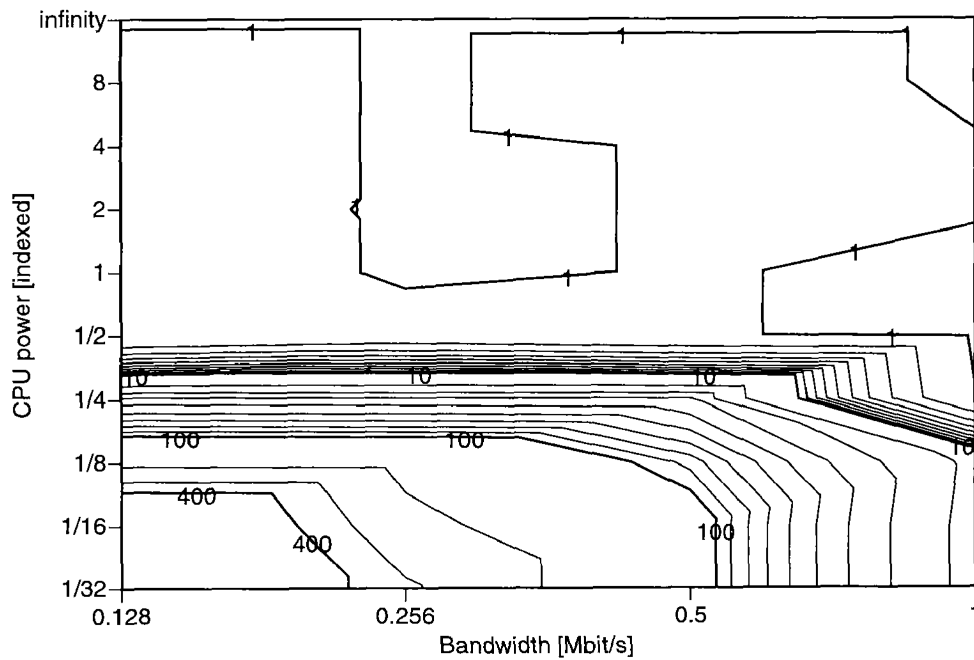


Figure 6.10: t_{diff} (in %) as a function of bandwidth and CPU power. Constant time limit.

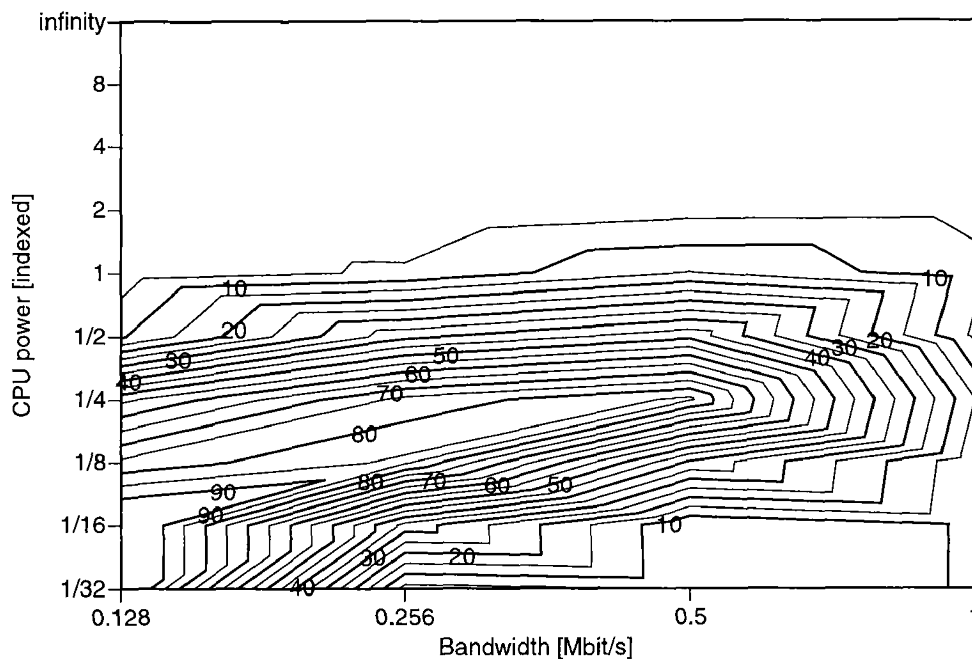


Figure 6.11: Performance loss (=100–bandwidth utilization) as a function of bandwidth and CPU power. Constant time limit.

more, the slower the CPU, the longer it takes to perform a given transformation and thus the more likely it is that the time limit is exceeded. Figure 6.10 shows the t_{diff}-metric as a function of bandwidth and CPU power in a contour plot. Note that both the x- and the y-axis are logarithmically scaled (base 2). The bandwidth levels of 0.128, 0.256, 0.5 and 1 Mbit/s reflect adaptation ratios α, as defined in Section 6.3.2, of 4.5, 17.7, 43.0 and 94.8% respectively. Note further that the z-axis is also logarithmically scaled (base 10; same number of contour lines for each order of magnitude). The results reveal no surprises, performance gets worse with decreasing bandwidths and with decreasing CPU power. We see that down to a CPU level of $1/2$ the time limit is not exceeded for all levels of bandwidths studied. This observation implies that if the CPU is fast enough for low bandwidths there won't be any problems at high bandwidths either (because the transformation costs decrease approximately with the same rate as the bandwidth increases). This finding also implies that the fixed costs indeed seem to be negligible for such a long time limit. This result confirms what was a speculation in Section 6.3.3. Furthermore, the figure indicates that the region with acceptable performance ($t_{diff} < 10\%$) seems sufficiently large to allow adaptation to be useful in a wide range of situations. There is obviously a sweet spot for CPU power (or transformation costs) below which performance dwindles rapidly. Note that for CPU power of less than $1/2$ the power of the 200MHz PC (i.e. for $y < 1/2$) a reduction of factor 2 in CPU power results in about a factor 2 increase in the time needed for the transfer (i.e., t_{diff} increases from $< 10\%$ to $\approx 100\%$).

Bandwidth utilization. Likewise, we would expect utilization of the available bandwidth to decrease either with decreasing bandwidth or decreasing CPU power, because the transformation costs go up. Figure 6.11 shows a contour plot of the performance loss (=100–bandwidth utilization) as a function of bandwidth and CPU power. Again, both the x- and the y-axis are logarithmically scaled (base 2). The z-axis is linearly scaled (a contour line for every 5% increase in performance loss). At first sight, the results seem to exhibit a non-intuitive behavior, as the performance loss does not increase continually with decreasing CPU power or decreasing bandwidth (in other words, the utilization does not decrease continually). This behavior can be explained as follows. Recall, that time is the primary optimization criterion for the feedback control loop, which tries to minimize $\mid t_{diff} \mid$. As a consequence, for a fixed level of bandwidth (e.g., 256 kbit/s), there is a level of CPU power ($1/4$ for this bandwidth), below which the sum of the transformation costs (to reduce the objects to minimal quality) and the transmission costs (to send these reduced objects) exceed the transmission costs of the uncompressed objects. In other words, it is cheaper "timewise" to deliver some objects (or all objects if the CPU is too slow) uncompressed than to reduce their quality and size first. Therefore, if the objects are sent uncompressed the utilization of the available bandwidth increases again (i.e. performance loss decreases). Once the CPU is so slow that all the objects are delivered uncompressed, the available bandwidth is completely utilized and the time limit is not exceeded further with decreasing CPU power because the time it takes to deliver the response is completely determined by the bandwidth, i.e. the time to transmit the objects. This behavior can be seen in Figures 6.10 and 6.11 (e.g., pick the bandwidth level of 0.5 Mbit/s and go along the axis indicating CPU power).

The fact that bandwidth utilization (Figure 6.11) first drops and then increases again with decreasing CPU power may seem to be at odds with the assumption of unidirectionality under-

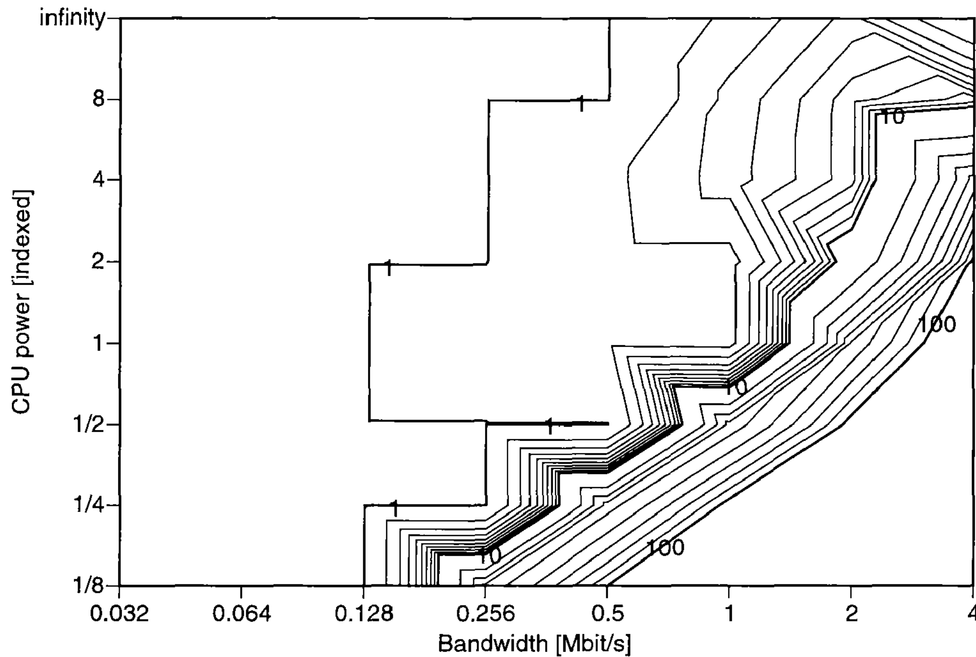


Figure 6.12: t_{diff} (in %) as a function of bandwidth and CPU power. Constant adaptation ratio.

lying the $2^k r$ experiment. There are two comments to make here. First, in the $2^k r$ experiment the levels of the factor *CPU* are varied from the low level (200 MHz Pentium Pro), which served as the basis for the scaling of the y-axis in the contour plots in Figures 6.10 and 6.11, towards faster CPUs (300 MHz Ultra Sparc and an infinitely fast CPU). In this range ($y \geq 1$) the assumption of unidirectionality is not violated. Thus, the conclusions drawn from the $2^k r$ experiment are valid. Second, note that the most important metric is t_{diff} (this is the metric our adaptation framework optimizes for) and note that the assumption of unidirectionality is not compromised for the t_{diff} metric.

In summary, the results reported in this section show that network-aware delivery with the goal to meet a time limit performs as expected, that is, it can provide predictable response times over a wide range of bandwidths and CPU powers. This conclusion may seem rather mundane, but it has to be emphasized that the observations on bandwidth utilization also imply that adaptation does not do anything bad, that is, does not perform worse than an appropriate static delivery policy would. (This result will be reconsidered under different aspects in Sections 6.5 and 6.6). Furthermore, we find that a medium performance workstation suffices to perform the communication-computation tradeoff required by applications such as Chariot for example. Moreover, we note that—unlike in the case of fixed adaptation ratios discussed in Sections 6.3.3 and 6.4.3—high bandwidths are not problematic per se (if the time limit is sufficiently coarse).

6.4.3 Results: Constant adaptation ratio

To investigate how the choice of the time limit effects performance we conduct an experiment similar to the one in the previous section. Instead of keeping the time limit fixed and thus varying the degree of adaptation required (α) with varying bandwidths, we now use a constant adaptation ratio $\alpha = 50$. Therefore, the higher the bandwidth, the smaller the time limit. The

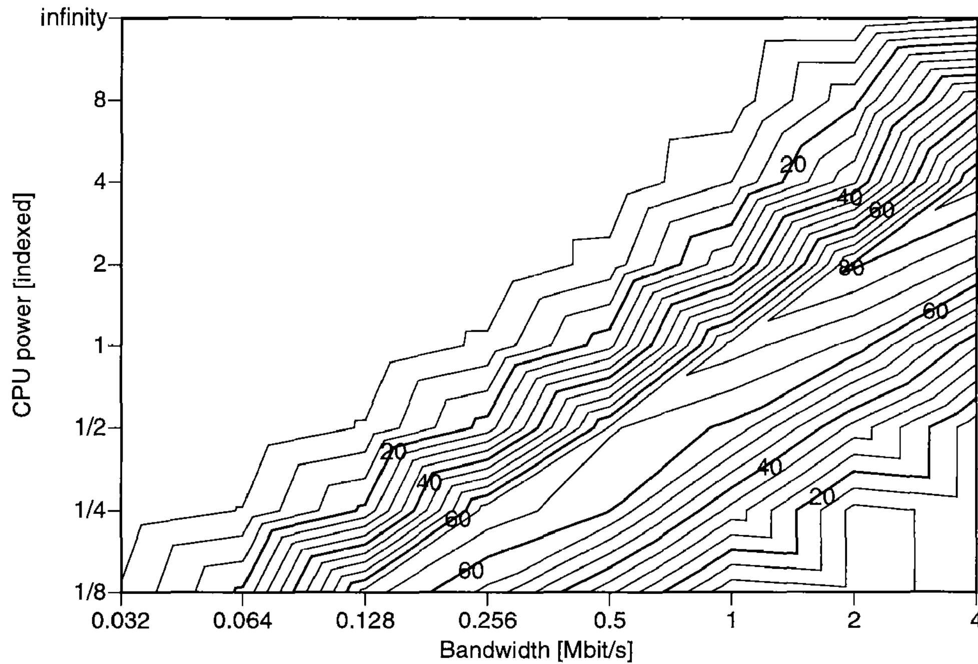


Figure 6.13: Performance loss as a function of bandwidth and CPU power. Constant adaptation ratio.

request used for this experiment requires 647 kB of data to be transmitted with $\alpha = 50$. Varying the bandwidth from 32 kbit/s to 4 Mbit/s means that the time limit decreases from 165 to 1.3 seconds. Figures 6.12 and 6.13 present the performance (metrics t_{diff} and performance loss) as a function of bandwidth and CPU power. Again, the x- and the y-axis are log-scaled (base 2) and the z-axis for the t_{diff} metric is also log-scaled (base 10). The y-axis is indexed and 1 indicates the effective transformation costs for JPEG compression on a 200 MHz Pentium Pro PC. The results are not surprising as there is a main trend showing that performance degrades either as bandwidth increases or CPU power decreases. Once the sweet spot of CPU power and bandwidth is reached performance seems to decrease linearly (with increasing bandwidth and decreasing CPU power). Increasing bandwidth and decreasing CPU power both imply a shift (towards smaller values) in the ratio of time available to transmit the data and time needed to carry out the transformations. The smaller this ratio, the more likely that the time limit is exceeded.

The effect of small time limits becomes obvious when studying the contour lines of the t_{diff} plot (Figure 6.12). Between $(bw = 0.256, cpu = 1/8)$ and $(bw = 2, cpu = 1)$ the contour lines form a straight line. Consider contour line for $t_{diff} = 40\%$ for instance. For $bw > 2$ we see that the contour lines break this linear trend (towards higher t_{diff} values, i.e. worse performance). This observation implies that the CPU must be proportionally faster for high bandwidths (and thus small time limits) than for small bandwidths to keep performance penalties small, that is, to amortize the fixed costs over the small time frame. Similar, however, less obvious results can also be derived from the performance loss plot in Figure 6.13. The fact that performance degrades more rapidly at higher levels of bandwidth (and smaller time limits) is reflected by a higher and a slightly broader "ridge" in the performance loss contour plot for $bw > 1$. The findings imply that for a time limit smaller than $\approx$ 3–4 seconds (reflecting a bandwidth larger than $\approx$ 1.5 Mbit/s in our experiment) the fixed costs (see Section 6.5) start to dominate the time

Factor	Effect	Variation	Conf. Interval
Mean	3.3		(3.2, 3.5)
AP	−2.1	12.5	(−2.2, −1.9)
AR	1.5	7.0	(1.4, 1.7)
BW BV	−1.8	9.3	(−1.9, −1.6)
Errors		15.5	($\sigma_e = 2.6$)

Table 6.9: No prepare required: Key factors, their interactions and their effects on t_{diff}. Factors listed if variation explained by factor $\geq 5\%$.

Factor	Effect	Variation	Conf. Interval
Mean	98.4		(98.2, 98.6)
RS	−1.2	6.8	(−1.3, −1.0)
CPU	1.0	5.1	(0.8, 1.2)
Errors		40.9	($\sigma_e = 3.2$)

Table 6.10: No prepare required: Key factors, their interactions and their effects on *bandwidth utilization*. Factors listed if variation explained by factor $\geq 5\%$.

available for transmission of the response. As alluded to in Section 6.3.3 there are a number of measures that may allow to reduce the fixed costs and thus may help to increase the range of time limits (towards smaller time limits) for which adaptation performs favorably. However, there will always be a lower limit on the time frame below which adaptation performs poorly. As a consequence, the type of network-aware content delivery mechanisms proposed in this dissertation may not be suited to meet very small time limits. Since this lower limit is clearly application-dependent, no final answer can be given as to what is the useful operation range (in terms of time limits) for network-aware delivery.

6.4.4 Computation-storage tradeoff

The factor *prepare required*, that is, whether images must be reduced at run-time or whether all the reduced versions of the images are precomputed has a significant impact on performance as indicated by the overall results presented in Section 6.3.3. The impact is significant enough to warrant a discussion whether the potential negative effects of on-line reduction in terms of missing time limits and underutilizing bandwidth are worth the savings in storage space compared to an approach which pre-computes and stores all the versions of the images.

Before drawing any conclusions we must first get a feeling for (i) the potential benefits of and (ii) the costs incurred by a solution which precomputes the responses or — more realistically — caches intermediate results. In the end, the question whether precomputation, or caching, yields any performance benefit boils down to the classic questions related to any kind of caching. This means that without application-domain knowledge, e.g., about user access patterns, the question cannot be answered conclusively. The following results provide some rough numbers on potential benefits and costs in the case of the Chariot system examined here.

Tables 6.9 and 6.10 show the results for those of the $2^k r$ experiments that rely on precomputed images. As can be seen, these experiments achieve almost perfect adaptation perfor-

Adaptation potential	Adaptation granularity	
	coarse	fine
low	2.4–2.7	6.4–6.7
high	2.9–3.3	7.0–7.8

Table 6.11: Ratio of storage space required for all precomputed versions of the images in a request to the space required to store the uncompressed versions only. Range spans the ratios observed for the requests C_{12}, V_{12}, C_{100} and V_{100}.

mance. On average, the time limit is exceeded by 3.3% and the bandwidth available is utilized to 98.4%. The factors that influence performance have only a very limited effect on average performance. In fact, the experimental errors explain most of the (small) variation in the measurements observed. This observation and the fact that *bandwidth* has no notable effect on performance (in contrast to the overall results presented in Section 6.3.3) indicate that network-aware delivery with precomputed images can tide networked users over a wide range of bandwidths and still meet even comparatively small time limits at satisfactory levels of quality. This result must be put in relation to the performance penalty that might have to be witnessed, if the images are reduced at run-time (as discussed in Section 6.4.2), and the costs incurred in terms of CPU usage and storage as discussed below.

Table 6.11 illustrates how much (i.e. how many times) more storage space would be required, if all the versions of images that can be requested were precomputed and stored on disk. The data reflects the properties of the requests C_{12}, V_{12}, C_{100} and V_{100} used in the $2^k r$ experiment and thus give only approximative figures. The amount of additional storage space required can be quite considerable (up to factor 8) and it increases with increasing adaptation granularity.

How to resolve the tradeoff between computation and storage remains to be evaluated in each particular application domain. Version caching has been implemented in the Chariot prototype, that is, in the instantiation of the framework (see [127] for details). Version caching would also be a useful addition to the general framework, however, to be implementable, an application-independent method of identifying "cache-hits" would have to be devised.

Apart from discussing the computation-storage tradeoff, it is interesting to note in Table 6.10 that the factors *request size* and *cpu power* turn up in the evaluation of bandwidth utilization. As noted above, the factors have an almost negligible effect. This result is good news, as the factor *request size* denotes a per object overhead. The fact that CPU effects the performance of experiments which rely on precomputed images shows that there are other sources of overhead (in addition to those discussed in the following section). Fortunately, these overheads are very small.

6.5 Costs of adaptation

So far, we have seen that adaptation can fulfill its objectives and may do so in a fairly wide range of situations. Hence, to gauge the claimed benefit of adaptation, we must contrast it to the (run-time) costs of the adaptation process. There are basically two sources of overhead to be considered: the decision making overhead incurred during the adapt phases (Section 6.5.1), and the CPU resources needed for the bandwidth tradeoff (Section 6.5.2). Again, we are interested

Factor	Effect	Variation	Conf. Interval
Mean	3.3		(3.3, 3.3)
BW	2.5	15.7	(2.5, 2.6)
AG	1.7	7.3	(1.7, 1.8)
PREP	−1.5	5.6	(−1.5, −1.5)
RS	1.5	5.5	(1.5, 1.4)
Errors		1.7	($\sigma_e = 0.9$)

Table 6.12: Relative adaptation (=decision making) overhead is represented as a percentage of the time used to deliver the response. Key factors, their interactions and their effects on the adaptation overhead are reported if variation explained by factor $\geq 5\%$.

in quantifying the effects of various external and internal factors on the adaptation costs. For that purpose, we reuse the evaluation methodology applied in Section 6.3; instead of considering t_{diff} or bandwidth utilization, we now use the decision making overhead and the transformation costs as the response variables of interest.

6.5.1 Adaptation overhead

The adaptation overhead captures the time spent on making adaptation decisions. Because in our design the adapt phases are not overlapped with "useful" (prepare or transmit) activity (see Section 4.4), the decision making overhead, if considerably large, may have a serious negative impact on adaptation performance. Table 6.12 shows the relative adaptation overhead and the key factors that influence it. Relative means that the numbers represent the decision making overhead as a percentage of the total time used for response delivery. On average, for all the $2^k r$ experiments only 3.3% of the time required to answer a request and deliver the response are consumed to make adaptation decisions. This result is rather promising because it indicates that the adaptation overhead is indeed fairly low and therefore confirms the statement made in Section 6.4.2, which claimed that adaptation does not incur higher overheads than a comparable static delivery policy. Studying the factors that effect the mean overhead reveals bandwidth as the primary factor. In principle, there may again be two reasons why bandwidth is the primary factor. First (as noted in Sections 6.3.3 and 6.4.3), high bandwidth implies small time limits which may be problematic as the adaptation costs cannot be amortized over the short time frames. Second, high bandwidths may effectively incur higher absolute costs (for whatever reason). Table 6.13, which reports the absolute adaptation costs, weakens the second argument as bandwidth does not (significantly) effect the absolute costs. As a consequence, adaptation costs (as are any other fixed costs) are problematic for scenarios with short time limits, and are otherwise mainly influenced by the factors *adaptation granularity*, *prepare required*, and *request size*.

High adaptation granularities and large requests negatively impact adaptation costs (both in relative and in absolute terms), because the "solution space" increases considerably with the adaptation granularity and the number of objects in the request. That is, there are considerably more combinations of objects and transformations possible that may achieve a particular response size and that have to be compared to maximize utility (see Tables 6.3 and 6.4). The negative impact of the factor *prepare required* can be explained as follows. If the objects are

Factor	Effect	Variation	Conf. Interval
Mean	0.78		(0.77, 0.79)
RS	0.74	21.6	(0.73, 0.75)
AG	0.44	7.8	(0.43, 0.46)
RS AG	0.42	7.1	(0.41, 0.44)
Errors		2.7	($\sigma_e = 0.29$)

Table 6.13: Absolute adaptation (=decision making) overhead [in seconds]. Key factors, their interactions and their effects on the adaptation overhead are reported if variation explained by factor $\geq 5\%$.

reduced at run-time, readjustments of the adaptation decisions may become necessary, e.g., if the transformation costs in terms of CPU usage (c_{prep}) or in terms of wallclock time (t_{prep}) are wrongly estimated. Such readjustments of the adaptation decisions obviously incur higher adaptation costs. In case the images are precomputed the likelihood that such readjustments become necessary is significantly lower.

6.5.2 Transformation costs

The number of CPU cycles required to reduce the quality of the objects transmitted is application-dependent. Therefore, we only briefly discuss (or review) three findings that may also be valid in a broader context.

First, analyzing the key factors effecting the total transformation costs of the $2^k r$ experiments (for a particular CPU), we find that the *relative* transformation costs are mainly effected by the factors bandwidth and degree of adaptation required. That is, the relative transformation costs increase with high bandwidths and high degrees of adaptation required. Following the argumentation of the previous sections this result appears obvious, as both factors shorten the time limit and thus the overhead for reducing the quality of the objects transmitted becomes large compared to the time limit.

Second, since the degrees of adaptation required are kept fixed (for varying bandwidths) in the $2^k r$ experiment, we would expect the *absolute* transformation costs to remain unchanged for varying bandwidths. However, we find that for the "slow CPU" (Pentium Pro) the absolute transformation costs are 14.2 seconds on average for the low bandwidth experiments and 11.3 seconds for the cases with high bandwidth. The observations on bandwidth utilization made in Section 6.4.2 also explain this result.

Third, the *absolute* transformation costs are effected only marginally by the degree of adaptation required—for the comparatively high degrees of adaptation required chosen in the $2^k r$ experiment ($\alpha = 10\%, 40\%$). This is somewhat surprising, since we would expect the transformation costs to increase (more drastically) with increasing degrees of adaptation required (i.e. with decreasing α). The result can be explained by the fact that the "constant" overheads for starting the transformation algorithm (e.g., JPEG compression) and for reading the object (image) from disk may outweigh the additional costs incurred by a higher compression ratio. These overheads are constant in the sense that they are not effected by the degree of adaptation required (beyond a certain value of α). The results in terms of transformation costs for the experiment described in Section 6.4.2 support this argument: Figure 6.14 plots the total absolute transformation costs as a function of bandwidth and the costs for a single transformation.

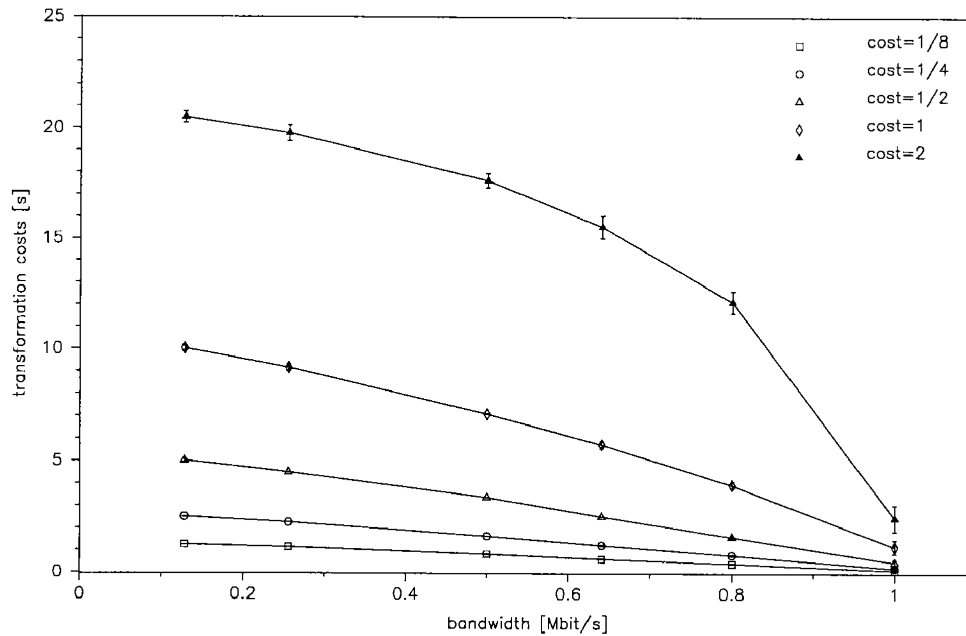


Figure 6.14: Total transformation costs for on-time request delivery as a function of bandwidth and the costs of a single transformation. Time limit for request = 30 seconds. Cost for a single transformation are scaled, 1 reflects the cost for JPEG compression on a i686/200 MHz PC.

The figure merely includes the results of experiments for which the time limit was exceeded by less than 2%. Recall that for this experiment (with a constant time limit), small bandwidths mean high degrees of adaptation required (i.e. low values for α). The figure shows that the total transformation costs increase towards smaller bandwidths. Note that although the absolute costs increase steadily, the size of the increase tends to get smaller with smaller bandwidths, that is, the cost curve flattens out towards smaller bandwidths. This observation supports the reasoning above as follows: For bandwidths larger than x ($x \approx 0.7$ Mbit/s in our example) the number of objects that need to be reduced increase as bandwidth decreases. For bandwidths smaller than x, all the objects in the request need to be reduced in quality and hence the degree to which individual objects must be reduced increases as bandwidth decreases. For the former cases (bandwidth $> x$) the increase in total costs (towards smaller bandwidths) is dominated by the costs incurred by the additional objects that need to be reduced. For the latter case, the fixed costs, that is, the overhead for starting the transformation algorithm and the costs for reading the object from disk, remain constant because all objects need to be reduced, and merely the variable costs for higher reduction ratios increase. If the fixed costs dominate the variable costs (as is the case in the Chariot system), the total transformation costs are indeed only marginally effected by the degree of adaptation required.

From the last finding, we can conclude that using an adaptation policy which concentrates quality reductions on as few objects as possible may be more advisable (to reduce transformation costs) than a policy which tries to distribute the quality reductions evenly among all the transmission objects (as noted in Section 6.3.3).

6.6 Utility of adaptation

As shown in the previous sections, network-aware delivery to meet a user-specified time limit is possible at fairly small costs in terms of adaptation overhead and in terms of transmission possibilities wasted (i.e. bandwidth utilization is high). The latter point is important when it comes to answering the question whether adaptation provides any "benefit" in addition to meeting the time limit, that is, whether adaptation is able to deliver results that are useful for an end-user. This usefulness (or utility) of a response generally captures some notion of *fidelity* [132] of a quality-reduced response with respect to the unchanged response. To assess whether adaptation is beneficial we must analyze the difference in *fidelity* of a *dynamically* produced response by a network-aware server and a *statically* defined response (e.g., optimized for users with low-bandwidth network access). As the notion of fidelity and hence the notion of "benefit" are clearly user- and application-dependent, it is important to understand how such a (presumed) benefit depends on the notion of fidelity, i.e. on the utility functions chosen.

In this section we show that network-aware delivery—in addition to being able to meet a user-specified time limit for a wide range of bandwidths—is able to provide superior utility compared to a static, non-adaptive delivery mechanism. However, the adaptation behavior seems to depend strongly on the utility function employed. In the context of the Chariot system, we discuss various notions of utility (Section 6.6.1) and conduct a simple experiment to validate our claims (Sections 6.6.2 and 6.6.3).

6.6.1 Various notions of utility

According to the service model used for the type of network-aware applications studied here (see Sections 3.2 and 3.3) there are basically two aspects that need to be considered as far as the utility of a particular response is concerned:

Object quality. How does the user value the quality of a single object? E.g., in the context of the Chariot system, how does the user-perceived quality of a JPEG image change with changing compression ratios (i.e. JPEG quality factors)? We describe the quality of an object i in the response by $q_i \in [0,1]$. q_i shall reflect the quality of object i (as perceived by the user) relative to its full quality version, thus, $q_i = 1$ means that the object is delivered unchanged, i.e. not reduced in quality.

Importance of an object. How does the user value (the quality of) a particular object in comparison with other objects in the request? In other words, how important is it that a particular object is delivered in high quality, for instance? The relevance of an object i is described by $r_i \in [0,1]$.

The utility of an individual object i can be defined as $u_i = q_i \cdot r_i$. The total utility of the response is then $u = \sum_{i=1}^{N} u_i = \sum_{i=1}^{N} q_i \cdot r_i$, where N is the number of objects requested. To simplify comparisons we normalize the utility function, such that the utility u_{max} of a non-reduced response is 1.[12] For both aspects effecting total utility there may be many highly user- and application-dependent interpretations of utility. In the context of the Chariot system, we

[12]Since q_i is 1 for non-reduced objects i, we merely need to normalize the relevance scores r_i, such that $\sum_{i=1}^{N} r_i = 1$ to obtain a normalized utility function.

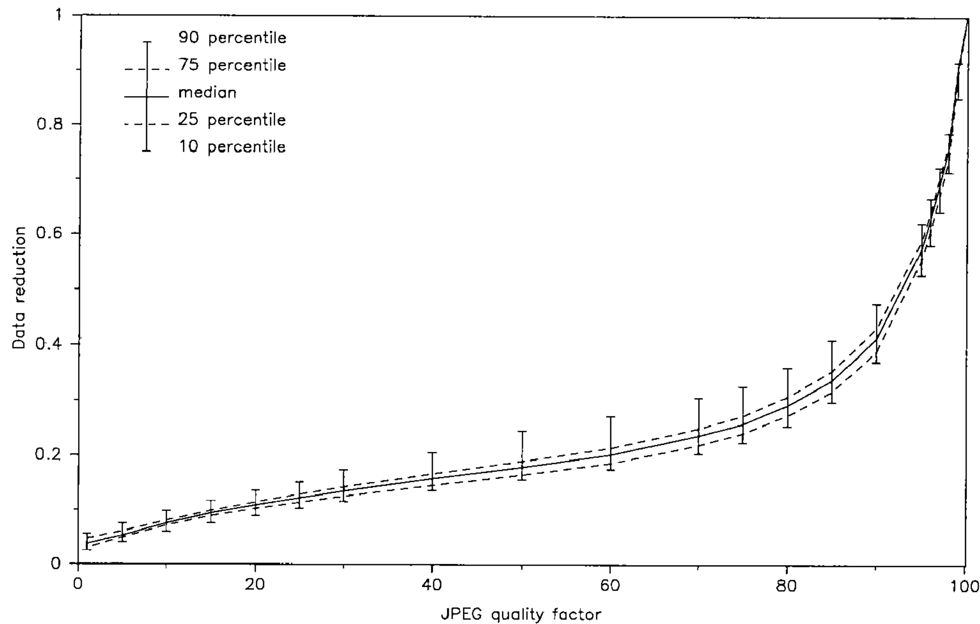


Figure 6.15: Size reduction achieved by JPEG compression as a function of the JPEG quality factor applied (sample set: images from requests C_{12}, V_{12}, C_{100} and V_{100}).

discuss three alternative interpretations for each of the aspects and show how they effect the adaptation behavior. (The quality definitions that are appropriate for a particular user may be selected at run-time.)

Object quality. As the size of a JPEG image (of a given resolution) is largely determined by the JPEG quality factor which was used to encode the image, it is obvious to characterize the fidelity of a JPEG image as a function of this quality factor. Figure 6.15 shows the relative size of a JPEG image as a function of the quality factor used to encode it. Hence, the first approach (denoted as Q_1) defines the quality of an object to be proportional to its size in bytes (convex curve in Figure 6.16). A second interpretation (Q_2), as chosen by Noble et al. [132], assigns the fidelity levels of JPEG images proportional to their quality factor as depicted in Figure 6.16 by the straight line. In contrast to these two fairly arbitrary choices of utility functions for image quality, a user study conducted by McIlhagga et al. [116] suggests that the fidelity of a JPEG image does only marginally degrade down to a quality factor of $\approx$ 30–40. Only quality factors smaller than 30–40 result in notably lower fidelity. Their observations are reflected in the concave utility function in Figure 6.16 (Q_3).

Importance of an object. The three interpretations taken to characterize the relevance of a single object in comparison with other objects in the request are the following: First, the user might consider all the N images to be equally relevant (variant denoted as R_1), that is, $r_i = 1/N$. This may be the case, for instance, if a user simply browses the image repository with no particular goal in mind. Second, the relevance of an individual image can be described as a function of a similarity measure[13] provided by the search engine (R_2: $r_i = f_1(similarity_i)$), e.g., $f_1(similarity_i) = \frac{similarity_i}{\sum_{k=1}^{N} similarity_k}$. The numerator is used to normalize the utility, such that the

[13]Section 6.6.2 gives a brief description of the two metrics "similarity" and "rank".

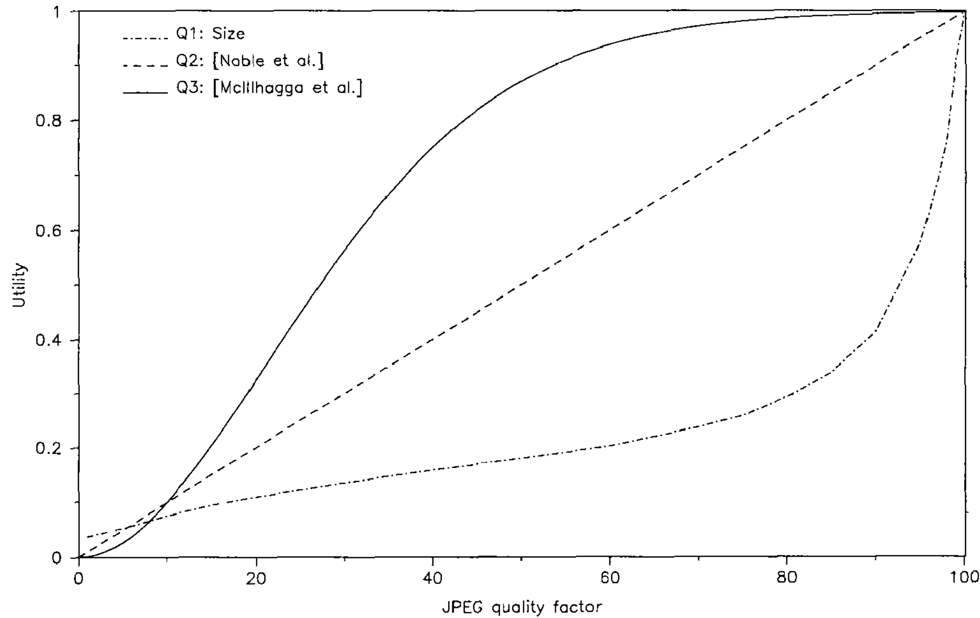


Figure 6.16: Various definitions of utility of a compressed JPEG image as a function of the JPEG quality factor applied.

total utility of a non-reduced response is 1. Third, ranks assigned by the search engine may serve as a means to strictly order the images in the request (R_3: $r_i = f_2(rank_i)$), e.g., an image i with $rank_i$ may be considered twice as relevant as an image j with $rank_j = rank_i + 1$, that is, $f_2(rank_i) = \frac{2^{-rank_i}}{\sum_{k=1}^{N} 2^{-rank_k}}$. Again, the numerator is used for normalization purposes only. The decrease by a factor of 2 is quite drastic. It is deliberately chosen to emphasize the effects of strong discrimination between the images.

6.6.2 Experiment

To study and compare the utility of non-adaptive and adaptive image delivery we conduct the following experiment within the Chariot system. A user issues a search for images that are similar to a given query image[14]. The search engine produces a ranked list of similar images and provides a similarity score ($\in [0,1]$) for each of the images. The similarity score for an image represents the probability that any randomly chosen pair of images from the database exhibits a weaker similarity than the query and result image [191]. The database contains the meta-data (feature data, etc.) of approximately 100'000 images [193]. Figure 6.17 depicts the normalized relevance scores for the sample query as a function of the rank and the interpretation of "object importance" (R_1, R_2, or R_3 as described in the previous section). Within 10 seconds the user wants to see 25 JPEG images with a minimal JPEG quality factor of 10. The images are stored at the server in the JPEG format (JPEG quality factor 100) and have a resolution of 360×240 pixels. The full quality images amount to 1.1 MB of data to be transferred. The two levels of bandwidth used for this experiment are 128 and 256 kbit/s.

We study five static (or non-adaptive) delivery policies and nine adaptive image delivery

[14]We use color features, enhanced with spatial information, for the similarity search. The methods used for feature extraction are described in [178, 40]. The query image and results are the same as in [178] (Figure 2).

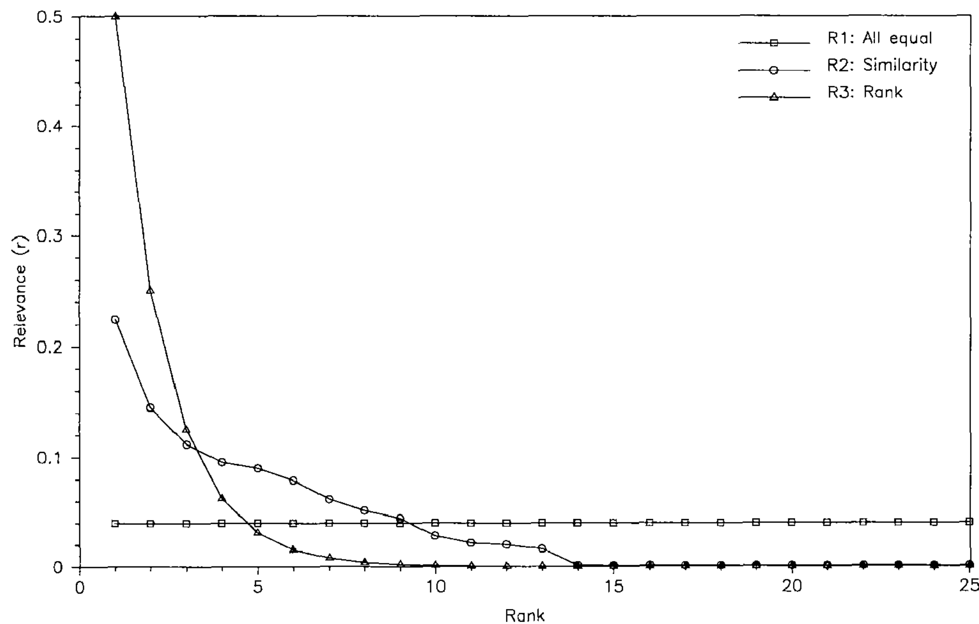


Figure 6.17: Normalized relevance scores r_i for the sample request (25 images) and the three notions of object importance (R_1, R_2, and R_3). Normalized means that the area below a curve is 1.

Object quality		Time [s]		Utility		
		128 kbit/s	256 kbit/s	Q_1	Q_2	Q_3
Low:	JPEG(10)	7.3	3.7	0.08	0.10	0.10
Medium:	JPEG(30)	*11.7*	5.9	0.13	0.30	0.56
Medium:	JPEG(50)	14.9	7.5	0.18	0.50	0.87
High:	JPEG(75)	20.6	*10.3*	0.26	0.75	0.98
High:	JPEG(100)	77.5	38.8	1.00	1.00	1.00

Table 6.14: Non-adaptive image delivery: Response time and total utility of response for various notions of object quality (time limit: 10 seconds). The total utility is not effected by notion of object relevance due to the normalization.

policies. The static policies deliver all images in the same quality. The images to be delivered are reduced to the appropriate quality off-line and stored in the version cache. The policy with lowest quality transmits the images encoded with the JPEG quality factor 10. The two policies delivering medium quality images use factors 30 and 50. And the two high-quality scenarios use factors 75 and 100. The adaptive image delivery policies of the network-aware server comprise the nine combinations of how to interpret the quality of an individual object (Q_1, Q_2 and Q_3) and how to interpret the importance of an object (R_1, R_2, and R_3). For each of these nine configurations we run two types of experiments, one with a fine adaptation granularity and one with a coarse adaptation granularity (Section 6.3.2). We would expect the utility to increase with finer adaptation granularity as there is more flexibility to find responses with a high utility.

Bandwidth	Adaptation granularity	All equal (R_1)			Similarity (R_2)			Rank (R_3)		
		Q_1	Q_2	Q_3	Q_1	Q_2	Q_3	Q_1	Q_2	Q_3
128 kbit/s	Coarse	0.12	0.26	0.41	0.28	0.57	0.74	0.59	0.73	0.97
	Fine	0.11	0.19	0.30	0.28	0.50	0.68	0.54	0.82	0.95
256 kbit/s	Coarse	0.27	0.69	0.95	0.69	0.88	0.99	0.97	0.99	1.00
	Fine	0.23	0.61	0.90	0.66	0.85	0.98	0.95	0.99	1.00

Table 6.15: Adaptive image delivery: Total utility of response for various notions of object quality and object relevance. Average response time: 10.17 s (standard deviation 0.13 s).

6.6.3 Results

Table 6.14 summarizes the experiments with non-adaptive image delivery. For each of the bandwidths there is at most one policy (JPEG(30) for the 128 kbit/s and JPEG(75) for the 256 kbit/s case) which matches the goal on response time considerably well ($t_{diff} \approx 10 - 15\%$). The high quality policy exceeds the time limit quite considerably and the low quality policy fails to exploit the time frame allotted. The last three columns report the utility of each of the responses according to three notions of object quality Q_1, Q_2, and Q_3 described in Section 6.6.1. Obviously, the total utility of a response is independent of the bandwidth available. Furthermore, the total utility of a response does not depend on the interpretation of the object importance because all the images are delivered in the same quality. Therefore the utility values listed in the table simply reflect the properties of all the utility functions shown in Figure 6.16.

Table 6.15 reports the results for the experiments with adaptive image delivery. On average, the 10-second time limit is exceeded by only 1.7%. To compare the utility of the delivered responses (e.g., with the result of a static policy) one must be cautious to only compare values of the same notion of object quality. E.g., comparing a value for Q_1 with a value for Q_2 is meaningless. There are three observations to make from these results.

First, we find that except for the R_1 experiments, the adaptive policies perform at least as well as the static policies and often outperform the static policies considerably in terms of utility. In scenarios where all the objects are of equal importance (i.e. the R_1 experiments) the adaptive policies achieve (slightly) lower utility as the two static policies whose response times approximately match the time allotted. The results for the R_1 experiments can be explained as follows: compared to a static delivery policy, adaptive image delivery incurs decision-making overheads. Even though these overheads are small in general (see Section 6.5), they still negatively affect bandwidth utilization. With the R_1 metric for relevance there are only limited opportunities for "smartness", that is, there are no opportunities for the adaptive policies to compensate the bandwidth loss experienced in terms of utility.

Second, we find that the utility seems to depend on the adaptation granularity. In particular, the utility seems to decrease with increasing granularity (except for the low bandwidth, R_3/Q_2 case). The reason why a finer granularity and hence a higher flexibility in decision making may not yield the expected improvement in terms of utility and may even result in lower utility is that the costs for adaptation increase with the adaptation granularity (see Section 6.5.1) and that these additional decision making overheads can outweigh the gains in utility. Furthermore, we note that the performance penalty witnessed by the experiments with fine adaptation granularity

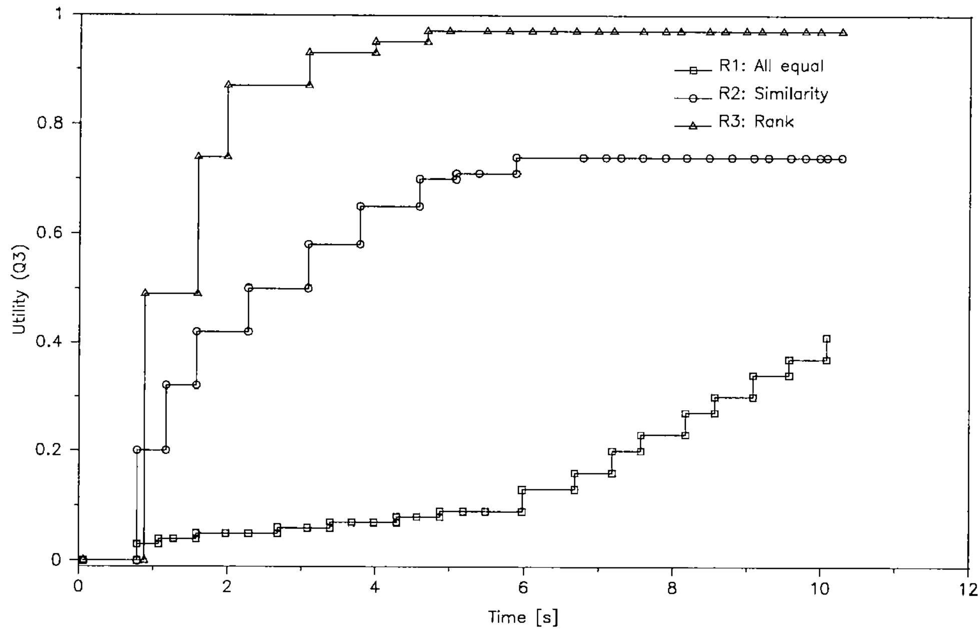


Figure 6.18: Utility progress curves for low bandwidth experiments with coarse adaptation granularity and the Q_3 utility function for object quality. The curves for the three relevance metrics R_1, R_2 and R_3 indicate that adaptation behavior depends strongly on the type of utility function used.

varies with the notion of utility employed. This observation can be explained by the varying costs to evaluate the utility functions and by the degree to which the use of a particular utility function allows the control loop to prune some parts of the solution space.

Third, the more the relevance scores discriminate between the objects in a request the better the adaptive policies perform compared to the static policies. If the relevance scores discriminate only weakly (or not at all, as in the R_1 case) between the objects in a request, the reductions in object quality are spread out fairly evenly among the images and thereby the network-aware image delivery operates similarly to a static policy that matches the time limit. On the other hand, if the relevance scores discriminate more sharply between the objects (as is the case for the R_3 experiments and less pronounced for the R_2 experiments) the quality reductions are passed onto the objects of least importance. Thus, compared to a static delivery policy which transfers all images in the same (e.g., medium) quality, a network-aware sender transmits the most important objects in full quality. The difference between the utility of the full quality and medium quality object is "amplified" most if the relevance scores are high for the important objects.

These observations are supported by the utility progress curves shown in Figures 6.18 and 6.19. The x-axis denotes time and the y-axis denotes the utility of the response. The utility function on object quality used for the figure is Q_3. The three curves of Figure 6.18 show how the total utility progresses for the relevance metrics R_1, R_2 and R_3 as the transfer of the objects in the response progresses. The length of a horizontal part in the curves represents the time required to transfer a particular image, and hence, since the bandwidth traces are constant, reflects the size and implicitly the quality of the image transferred. Vertical lines in the curves depict the utility gain achieved by the image transferred. The utility at the end of the transfer reflects the values for low-bandwidth, coarse adaptation granularity and utility function Q_3 listed

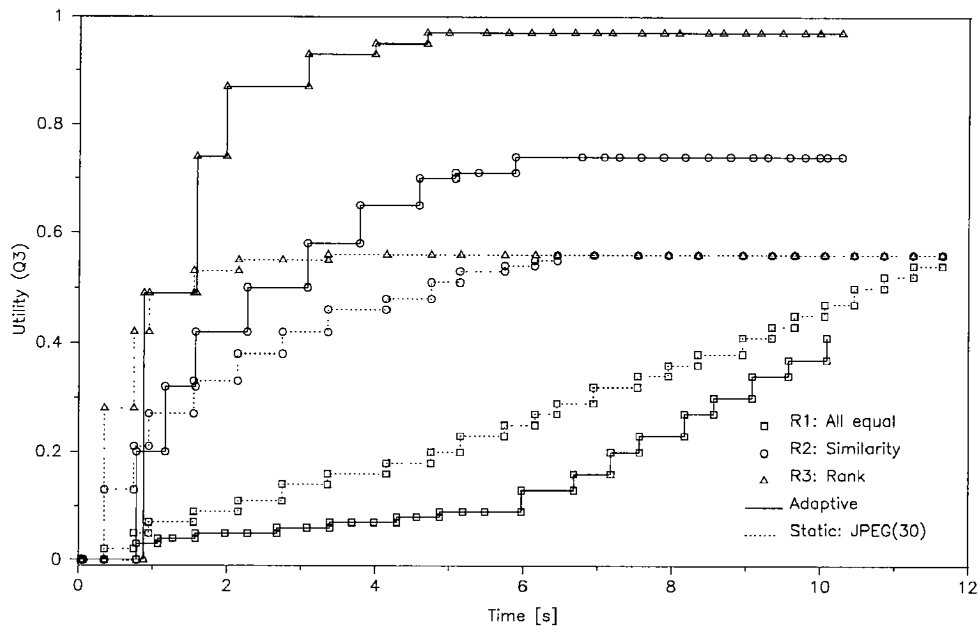


Figure 6.19: Utility progress curves for the adaptive low bandwidth experiments with coarse adaptation granularity and the Q_3 utility function. The comparison with the respective utility progress curves for non-adaptive (JPEG(30)) image delivery reveals that the more discriminative the relevance score is ($R_3 > R_2 > R_1$), the better the adaptive delivery mechanisms perform.

in Table 6.15.

Figure 6.19 repeats the results from Figure 6.18 for the adaptive image delivery and adds three utility progress curves for the non-adaptive experiments for JPEG(30). The non-adaptive image transfers all achieve the same utility (as reported in Table 6.14). As can be seen from the figures, the R_1 curves progress steadily. The adaptive and non-adaptive delivery attain comparable utility after 10 seconds[15]. Due to the stronger discrimination between the objects, both the R_2 and R_3 experiments start transmitting higher quality objects than the respective experiments with non-adaptive image transfer policies and are therefore able to achieve higher utility.

In summary, we find that adaptation can be beneficial because it is able to deliver the response within a user-specified time limit, and because adaptation can be smart about which of the objects delivered must be reduced in quality to attain the goal on time and to maximize the utility of the response. Furthermore, we find that the adaptation behavior varies considerably with differing notions of utility. We also note that our framework and in particular the heuristic approach taken to arrive at adaptation decisions, has proven to be flexible enough to deal with a variety of utility functions for both object quality and the importance of objects (and can therefore also be reused by different applications). As far as the efficacy of the various utility functions is concerned, we emphasize that this dissertation merely presents a framework for experimentation but takes no position on what utility functions should be used. More work is needed to define appropriate utility functions, i.e., utility functions that are meaningful to users in a particular application domain. To pursue this goal, e.g., in the context of Chariot, we need to (better) understand user concerns: What is relevant to a user when searching for similar

[15]The knee in the R_1 curve for the adaptive experiment is a direct consequence of the coarse adaptation granularity used, which provides only the two JPEG quality factors 50 and 10 at low image quality (Section 6.3.2).

images? How do reductions in image quality (achieved by the various compression schemes) relate to reductions in utility? As far as the second issue is concerned, the user study performed by McIlhagga et al. [116] may serve as a start. Our framework provides a flexible platform for further experimentation on these issues.

6.7 Accuracy of resource models

As shown in the previous sections, the type of network-aware delivery presented in this dissertation achieves encouraging results. Recall that the adaptation depends on several models for (network and end-system) resource availability and resource consumption (cost and size models for transformations). To reduce complexity, the experiments evaluated so far have been conducted using simple resource models that provide timely and reasonably accurate information (see Section 6.1.3).

The question that remained unanswered so far is: How sensitive is model-based adaptation to the accuracy of the resource models? The effectiveness of adaptation will depend on two factors. A first factor is the delay of the feedback loop: the shorter the delay, the better the application will be able to track changes in the network for instance. The second factor is the accuracy of the feedback information. We suspect that inaccurate information can degrade performance by having the application adapt needlessly, or by having the application select an incorrect operating point.

This section discusses the questions of timeliness and accuracy of the feedback signals in the context of the models for network resource availability. We use bandwidth estimation as the example to illustrate the concerns raised by inaccurate resource models. We concentrate on the problem of bandwidth estimation because it seems to be the most elusive. Models for resource consumption, that is, cost and size models for the transformations, are application-specific and may be quite easily obtainable (see [189] in the case of Chariot). Accurate host load estimation is certainly non-trivial [41, 42], however, it is easily avoidable if using a dedicated server. As far as bandwidth estimation is concerned, there is currently no mechanism in place to query the performance (bandwidth) of an end-to-end Internet connection, and as can be judged from the wealth of literature tackling the problem of dynamic bandwidth estimation (e.g., [29, 104, 141, 94, 18, 3, 44]) such estimations do not seem to be all that easily achievable.

Even though we study only the effects of inaccurate bandwidth estimation, we see no reason why the conclusion drawn in this section should not carry over to the other resource models used in our framework.

6.7.1 Experiments

We suspect that performance, that is, the framework's ability to meet the time limit and to achieve high bandwidth utilization, degrades considerably with increasing untimeliness and inaccuracy of the bandwidth estimates.

We conduct a first simple experiment to study the effects of untimely, that is, delayed, feedback (Section 6.7.2). We run an experiment similar to the experiment described in Section 6.2 and introduce varying degrees of delay into the feedback loop. The details of the experimental setup are given in [17] ("Internet experiment").

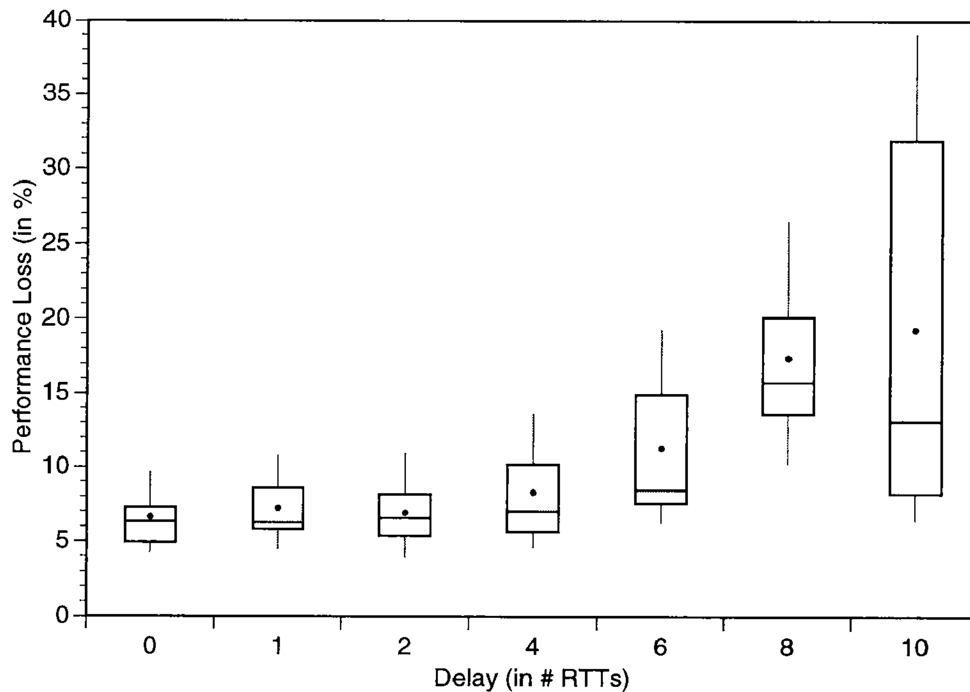


Figure 6.20: Box-plot of performance loss as function of delay in obtaining bandwidth information.

To study the effects of inaccurate bandwidth estimates we conduct two additional experiments. In a first step (Section 6.7.3), we repeat the experiment described in Section 6.4.3 (at a CPU level of 1). We study the adaptation performance as a function of bandwidth and the factor by which the available bandwidth is under- or over-estimated. Again, we use constant bandwidth traces in these experiments and keep the adaptation ratio constant with varying bandwidths. In a second step, we study the effects of under- and over-estimating bandwidth in real-world situations exhibiting highly fluctuating bandwidths (Section 6.7.4).

6.7.2 Timeliness

To study the effect of untimely bandwidth information, we run the "Internet" experiment described in [17] varying the amount of delay introduced in the bandwidth estimates. A delay of n round-trip times (RTT) means that the application learns about the currently available bandwidth only after n RTTs[16]. We investigate different levels of delay: 0 (which is the reference behavior as reported in [17]), 1, 2, 4, 6, 8, and 10 round-trip times; the average RTT for the transfer was approximately 100 ms. For each level, we conduct 40 experiments under the same conditions as the example above and summarize the performance penalty introduced by the delayed bandwidth information in the box plot in Figure 6.20. The x-axis shows the level of delay, the y-axis represents the relative performance loss (= 100−bandwidth utilization) experienced by the adaptation process. The top and bottom line, and the line through the middle of the box correspond to the 3rd quartile, 1st quartile, and median respectively; the whiskers on the bottom and top extend from the 10th percentile to the 90th percentile; the dot indicates the mean.

[16]Delayed bandwidth information *may* be inaccurate (e.g., if bandwidth changed within the n RTTs), but *need not* be inaccurate (e.g., if bandwidth remained unchanged).

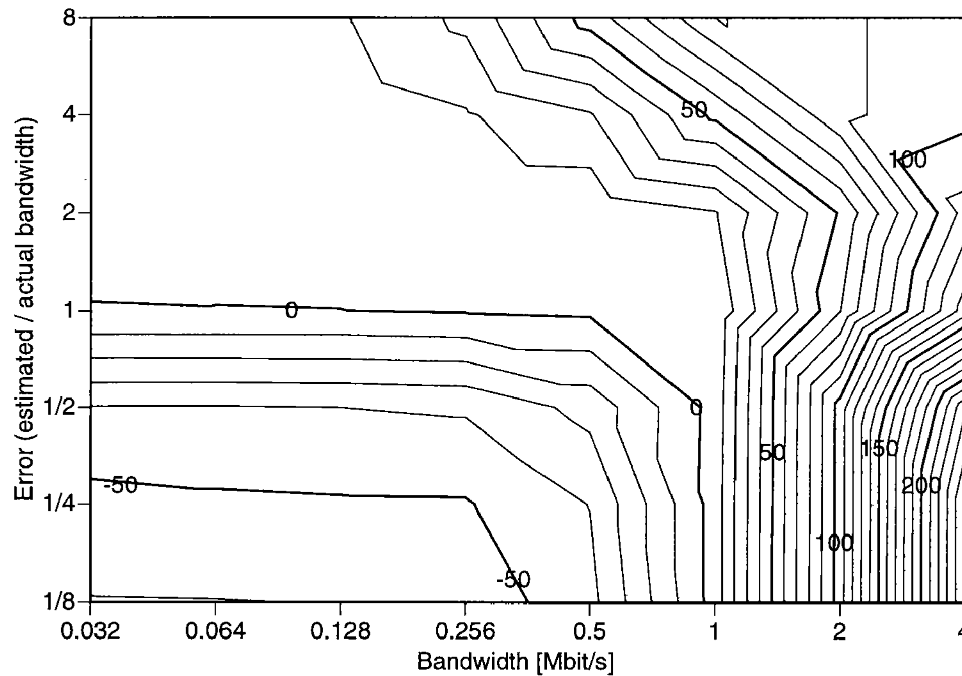


Figure 6.21: Constant adaptation ratio: t_{diff} (in %) as a function of bandwidth and inaccuracy of bandwidth estimation. The inaccuracy of the bandwidth estimation is characterized by the ratio of estimated and actually emulated bandwidth.

The figure shows that out-dated information on the bandwidth available does *not necessarily* result in lower performance; the 10th percentiles of the experiments with 0–6 and 10 RTTs delay are all in the range of 6–9 %. However, untimely and hence potentially incorrect estimates on the bandwidth available *can* increase the number of situations that require adaptation, and therefore can increase the overhead of the decision making. Furthermore, the more often adaptation is required, the higher the risk that we encounter limited adaptation potential or problems with the scheduling of the *prepare* and *transmit* phases. A high number of situations that call for adaptation may eventually result in a higher penalty in terms of lost opportunities to transmit. Untimely information therefore *can* increase the risk of performance loss, as is illustrated by the data for both the 3rd quartile and the 90th percentile. These values increase considerably with delayed bandwidth information.

The results indicate that the model-based adaptation is not very sensitive to (small) delays in the feedback loop. On the other hand, the results show that considerable performance penalties may have to be witnessed as the bandwidth estimates become less timely.

6.7.3 Accuracy: Constant bandwidth

There is no standard way to quantify the accuracy of a model, and common metrics, such as the sum of the square errors (SSE) [82], are not particularly intuitive and may even disguise some effects. E.g., with the SSE-metric, situations where the bandwidth available is under-estimated by a constant amount cannot be distinguished from situations where the bandwidth is over-estimated by the same amount. Therefore, we use a simple metric for the inaccuracy of the bandwidth estimates, that is, the factor by which the bandwidth actually available is

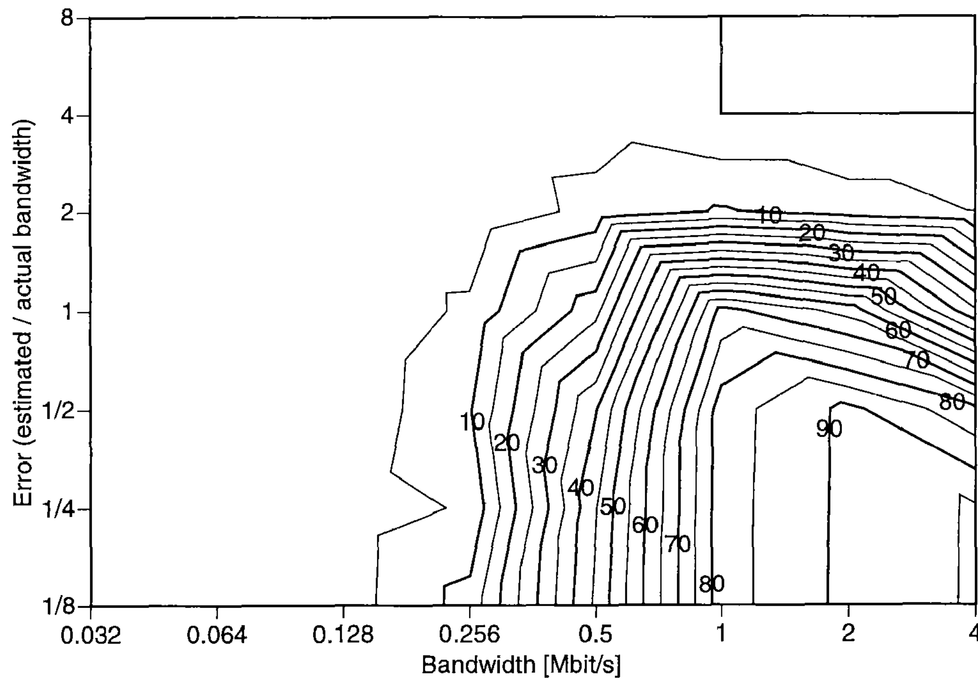


Figure 6.22: Constant adaptation ratio: Performance loss (= 100−bandwidth utilization) as a function of bandwidth and inaccuracy of bandwidth estimation.

(consistently) over- or under-estimated.

The contour plots in Figures 6.21 and 6.22 show the metrics t_{diff} and performance loss (= 100−bandwidth utilization) for the experiment described in Section 6.4.3 as a function of the bandwidth available (x-axis) and the factor by which this bandwidth is under- or over-estimated (y-axis). $y < 1$ reflects under-estimation, $y > 1$ denotes over-estimation of the bandwidth available. The results for $y = 1$ are identical to the results in Figures 6.12 and 6.13 for the CPU level of 1.

Under-estimation of the bandwidth available results in the transfer finishing considerably ahead of time. The percentage by which the time limit is under-utilized levels off at a certain degree of under-estimation. Under-estimation of bandwidth results in t_{needed} being over-estimated and hence results in unnecessarily high reduction/compression ratios being applied. Only after having transmitted the reduced objects can the control loop learn that the adaptation was too conservative. Since the bandwidth is consistently under-estimated the process repeats. Even if the control loop would recognize the consistent under-estimation, it may be too late to react because there may no longer be enough potential to "re-expand" the objects, and even if the time limit could be met, only the less relevant objects may benefit from such "re-expansion". The under-utilization of the time frame allotted levels off because there is a limit on how much the control loop is allowed to reduce the objects in quality. At high bandwidths (and hence short time limits), the negative impact of under-estimating the bandwidth seems to be compensated and outweighed by the effects described in Section 6.4.3. However, bandwidth utilization suffers, because the conservative adaptation decisions that are provoked by the far too low bandwidth estimates aggravate the problems discussed in Section 6.4.3.

Over-estimation of the bandwidth available seems to have no negative impact on bandwidth utilization and may even lead to improvements as Figure 6.22 suggests. Over-estimating the

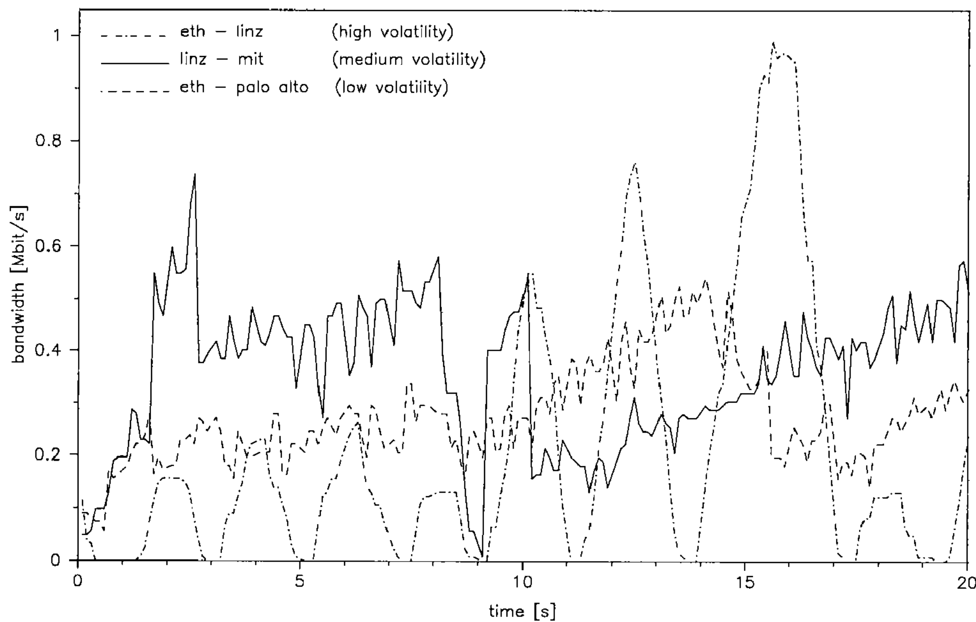


Figure 6.23: Bandwidth traces of three Internet transfers exhibiting considerable volatility.

bandwidth results in under-estimation of t_{needed} and thus leads to over-optimistic adaptation decisions that have to be readjusted later on. Therefore, consistent over-estimation results in the time limit being exceeded. The smaller the flexibility for readjustments, e.g., the smaller the time limit, or the smaller the reduction potential, the more likely it becomes that the time limit is exceeded. This effect can be observed in Figure 6.21. The same effect is to be expected if the number of objects requested is considerably smaller.

In summary, we find that model-based network-aware applications are not highly sensitive to small degrees of inaccuracies in the bandwidth estimates. However, we also find that inaccurate bandwidth estimation (if considerably inaccurate) may be problematic for the performance of network-aware delivery—even for scenarios with constant bandwidths. Over-estimation of the bandwidth available must be quite dramatic before performance implications can be observed.

6.7.4 Accuracy: Volatile bandwidth

To study the sensitivity of network-aware delivery to the accuracy of the bandwidth estimation in more realistic scenarios, we conduct a similar experiment as in the previous section. Instead of using constant bandwidth traces we use three traces picked out of a set of loss-afflicted TCP connections obtained from a large-scale Internet experiment [19, 18] (see also Chapter 8). To choose the bandwidth traces, we cluster all the (4148 TCP Reno) connections into three different groups according to the volatility of the bandwidth trace. We characterize volatility by the ratio of standard deviation and mean bandwidth. From each of the three groups we select one trace at random. Figure 6.23 shows the three traces used for the experiment. The trace "ETH–Linz" represents the high, "Linz–MIT" the medium and "ETH–Palo Alto" the low volatility scenario. Although the bandwidth traces exhibit a wide range of volatility, the respective connections achieved approximately the same average bandwidth. Apart from the bandwidth traces the request and the experimental setup are the same as used in the experiments

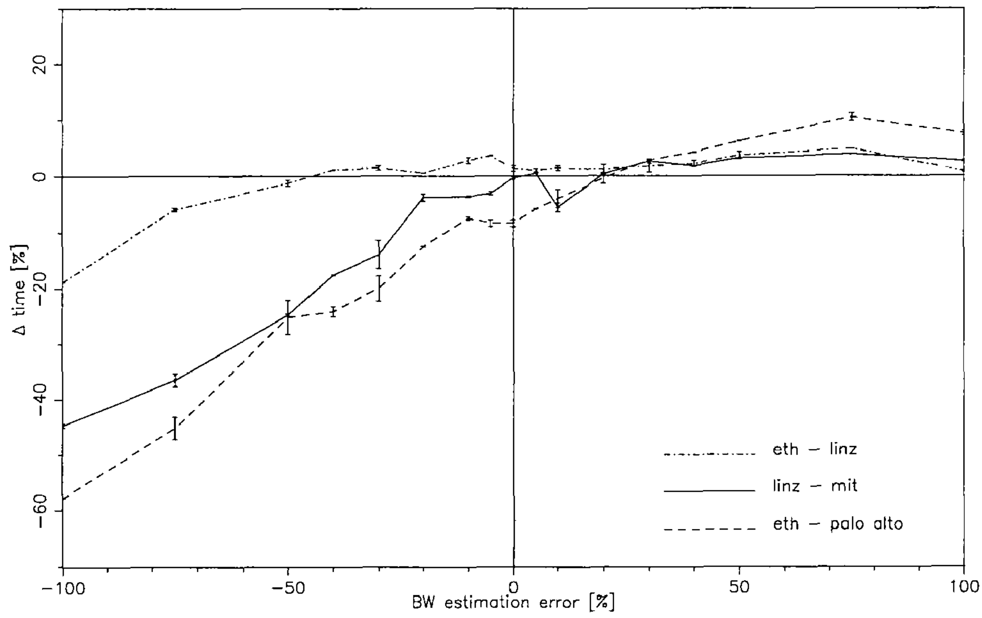


Figure 6.24: t_{diff} as a function of inaccuracy of bandwidth estimation.

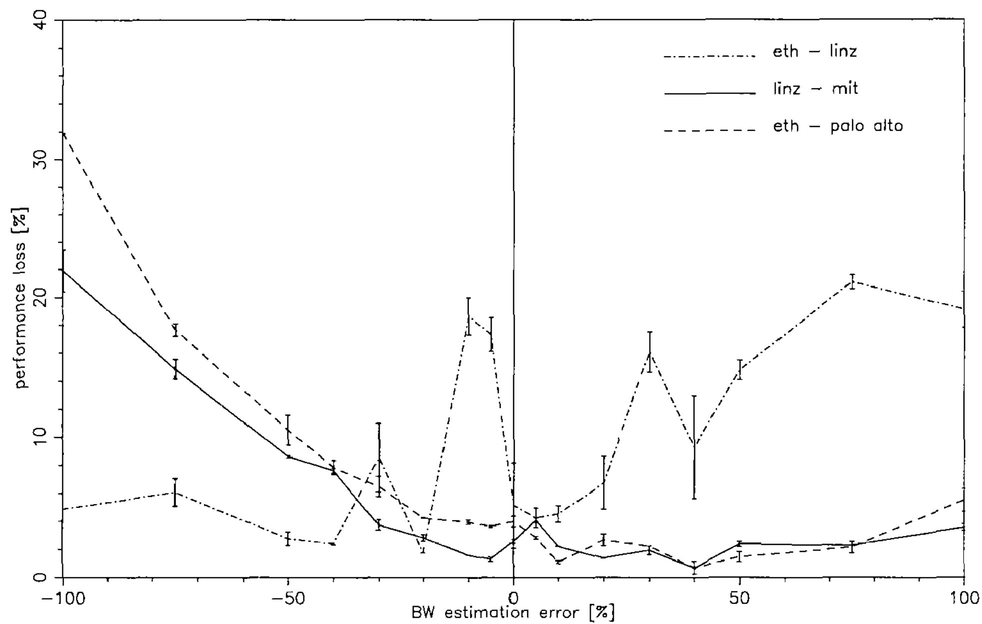


Figure 6.25: Bandwidth utilization as a function of inaccuracy of bandwidth estimation.

described in Section 6.2. The time limit to deliver the 25 JPEG images is set to 15 seconds. The experiments are repeated 5 times.

Figures 6.24 and 6.25 plot the average performance (t_{diff} and performance loss) as a function of the error e introduced in the bandwidth estimates; $e = 100 \cdot (estimated - actual)/actual$. The error bars depict the confidence interval for the mean at a 95% confidence level. Note, $e = -100\%$ means that the bandwidth is estimated to be 0. Likewise, $e = 100\%$ reflects the fact that the bandwidth is consistently over-estimated by a factor 2. The figures support the general trend noted in the constant bandwidth experiments. In particular in the cases with low and medium bandwidth volatility, we find that under-estimation of the bandwidth available results in the transfers finishing considerably ahead of time and bandwidth utilization suffering badly due to the overly conservative adaptation decisions. On the other hand, over-estimation does not seem to have much of a negative impact, neither timewise nor in terms of bandwidth utilization. To the contrary, it may even lead to improved performance compared to experiments conducted with $e = 0\%$. However, caution must be applied when interpreting these results (that is, it may be unwise to conclude from these results that bandwidth over-estimation is not problematic). If the time limit is shorter and/or the number of objects requested is smaller, then the flexibility to react to the effects of consistent over-estimation will decrease rapidly and so will the performance.

Considering the experiments with high bandwidth volatility we see quite different effects, in particular as far as bandwidth utilization is concerned. The curve named "ETH–Linz" in Figure 6.25 indicates that inaccurate bandwidth estimates are fairly problematic in situations with highly fluctuating bandwidths. Bandwidth utilization seems to be quite sensitive to even small changes in the accuracy of the estimates. The error bars are wider than for the less volatile bandwidth traces. Although no clear trend is discernible how utilization develops with increasing inaccuracy of the bandwidth estimates, it is important to note that performance problems are likely to arise if the bandwidth estimator performs badly.

In summary, we find that the effects of inaccurate bandwidth estimates depend on the volatility of the bandwidth available. Model-based network-aware delivery is not very sensitive to the accuracy of bandwidth estimates for low and medium levels of bandwidth volatility. This observation confirms the results from the $2^k r$ experiment in Section 6.3.3, which showed that performance is not effected for the degrees of bandwidth volatility present in those experiments.

However, significant performance penalties may have to be witnessed if bandwidth estimates are inaccurate. This problem seems to be exacerbated in situations with highly fluctuating bandwidths. As a consequence, network-aware applications can benefit considerably from timely and accurate bandwidth estimators. How accurate would such an estimator have to be? The results from the experiments using Internet traces seem to suggest that an accuracy of $\pm 10 - 20\%$ should suffice to stay within $\pm 10\%$ of the time limit. Clearly, more work is needed to understand the impact of inaccurate bandwidth estimates on the performance of network-aware applications and to draw final conclusions on how accurate an estimator should be.

6.8 Summary

This chapter takes a systematic approach at addressing the three basic concerns in the evaluation of network-aware applications.

When does adaptation work? We find that the model-based adaptation proposed in this dissertation is able to fulfill its goals, i.e. adaptation is able to meet a user-imposed time limit while achieving high bandwidth utilization, which is a prerequisite for achieving high quality responses. The detailed evaluation of the key factors effecting adaptation performance reveals that the adaptation mechanisms are fairly robust in delivering high quality responses regardless of most of the factors studied in the experiments and that performance considerations boil down to the *communication-computation* tradeoff striven for by network-aware content delivery. In particular, we observe that adaptation performance is fairly robust and hardly affected by bandwidth volatility (for low and medium levels of bandwidth volatility). Furthermore, we find that network-aware delivery can provide predictable response times over a wide range of bandwidths and CPU powers.

Two issues can hamper the effectiveness of network-aware delivery. First, the performance of dynamic adaptation is mainly limited by CPU bandwidth. As a result, we find that (i) network-aware content delivery may not be suited to meet very small time limits and (ii) that the caching of intermediate results, i.e. quality-reduced objects, can provide substantial benefits. Note, however, even in scenarios where adaptation fails to provide the service expected because of CPU limitations, adaptation does not perform worse than an appropriate static delivery policy would. Second, although the control loop is not overly sensitive to the accuracy of the resource models, and in particular to the accuracy of the bandwidth estimates, inaccurate information can have a significant negative impact on performance. The performance problems that inaccurate bandwidth estimates can cause are aggravated if bandwidth fluctuates heavily. As a result, we conclude that timely and accurate bandwidth estimation is important to the efficacy of network-aware applications.

How much does adaptation cost? Decision-making overheads turn out to be fairly small ($< 5\%$). They mainly depend on the adaptation granularity and the number of requested objects. Transformation overheads can be reduced by appropriate adaptation policies.

Is adaptation beneficial? Compared to a static delivery policy adaptation can provide a benefit to the user because it is able to deliver the response within a user-specified time limit and because it can be smart about which of the objects delivered must be reduced in quality to attain the goal on time and to maximize the utility of the response. We note that adaptation behavior varies considerably with differing notions of utility and although our work takes no position on what utility functions should be used and simply provides a framework for experimentation, we find that the framework has proven to be flexible enough to deal with a wide range of utility functions for both the object quality and the importance of objects.

Chapter 7

Dynamic bandwidth estimation

7.1 Introduction

Network-aware applications must address the following two questions (Section 3.4): first, how can we find out about (dynamic changes in) the bandwidth available on the path from the sender to the receiver, and second, how can an application adapt its delivery process (to such dynamic changes) to meet the time limit. The treatment of these two issues has been separated. Chapters 4–6 detailed the framework that answers the second question. One of the main conclusions drawn from the evaluation of the framework in Chapter 6 is that timely and accurate bandwidth estimation is important to the efficacy of network-aware applications. Ideally, to make sound adaptation decisions network-aware applications require accurate *predictions* about the availability of bandwidth in the near future (see Chapter 4).

The following chapters address the first question and discuss solutions for the problems how to find out about available bandwidth, and how to detect dynamic bandwidth changes. This chapter is organized as follows. Section 7.1 defines the term *available bandwidth* and provides a rough classification of the different approaches for dynamic bandwidth estimation. Section 7.2 reviews related work according to the classification presented. Section 7.3 gives a qualitative comparison of the various approaches and identifies open issues to be addressed in the subsequent chapters.

7.1.1 Available bandwidth

We can distinguish two different application-relevant characteristics as far as bandwidth information is concerned: *bottleneck* and *available* bandwidth [143, 141]. The bottleneck bandwidth gives an upper bound on how fast an application *can possibly* transmit (given the characteristics of the links traversed on the path from sender to receiver). In contrast, the term available bandwidth is less well-defined. It gives an estimate on how fast the connection *should* transmit to preserve network stability. (Network stability is a primary issue for congestion control mechanisms.) Thus, available bandwidth never exceeds bottleneck bandwidth and in fact can be much smaller (if there is considerable contention for bandwidth).

While knowledge about the bottleneck bandwidth is useful in bounding the approximations for the bandwidth estimates, information about the dynamics of the bandwidth available to an individual connection is the metric of importance to network-aware applications. In the

best-effort Internet, the bandwidth available to an individual flow (connection) can be approximated as a function of the bottleneck bandwidth and the aggregate bandwidth consumed by all competing flows at the bottleneck element (router or link) of the network path from sender to receiver[1]. In addition, since different (congestion-aware) protocols use the available bandwidth differently, the bandwidth experienced by the application also depends on the transport protocol used.

The IPPM (IP Performance Metrics) working group of the IETF refers to available bandwidth as follows [102] (and this is the definition we adopt): *Bulk transport capacity (BTC)* is a measure of a network's ability to transfer significant quantities of data with a single congestion-aware transport connection (e.g., TCP). The intuitive definition of BTC is the expected long term average data rate of a single TCP connection over the path in question. Central to the notion of bulk transport capacity is the idea that all transport protocols should have similar responses to congestion in the Internet (cf. Section 2.1). Because there are many congestion control algorithms (and hence transport protocol implementations) permitted by IETF standards, the notion of available bandwidth will be dependent on the transport protocol in use [102].

While there is agreement over the definition of a metric for available bandwidth that is useful to bulk-transfer applications [28, 102, 174] such as the network-aware applications considered here, different approaches to measuring and estimating available bandwidth exist. This chapter classifies and discusses the numerous different approaches proposed in the literature.

7.1.2 Discovering and monitoring bandwidth

Network-aware applications that adapt to both network heterogeneity *and* bandwidth fluctuations are confronted with two problems as far as dynamic bandwidth estimation is concerned.

Bandwidth discovery. As discussed in Section 4.7 network-aware applications are faced with the start-up problem. The problem is that the application must find out about the available bandwidth quickly, first, to be able to adapt to network heterogeneity, and second, to neither waste bandwidth nor risk missing the deadline. Ideally, applications would like to know about available bandwidth in advance, that is, before the application starts delivering data. We use the term *bandwidth discovery* to refer to the activities related to bandwidth estimation prior to data transmission.

Bandwidth monitoring. To learn about bandwidth fluctuations during a data transfer, the bandwidth available to a network-aware application must continually be *monitored.* The estimation of future bandwidth availability must take place at run-time. Here, an important concern is that monitoring should be timely enough so as not to hamper application agility (see Section 4.8). That is, a bandwidth monitor should be able to produce new bandwidth estimates at least at the rate at which the applications make new adaptation decisions.

Our main focus lies on bandwidth monitoring and on-line bandwidth estimation (and not on bandwidth discovery) for two reasons. First, such a focus is consistent with our work on

[1]This statement is only valid under the simplifying assumption that the flow is not bottlenecked by the sender or the receiver. That is, the sender has always data to send, and the connection is not limited by a slow receiver.

the framework for network-aware applications, which mainly concentrated on steady-state behavior, and which treated start-up related problems as a secondary issue. Second, compared to the related work in bandwidth discovery, bandwidth monitoring has received considerably less attention in the literature.

7.1.3 Bandwidth modelling and prediction

In contrast to other performance metrics such as the round-trip time, available bandwidth is not a simple metric that can be assessed with an instantaneous measurement. There exist many techniques to estimate available bandwidth. We distinguish the following two orthogonal issues:

Bandwidth modelling. There are a number of different mechanisms for estimating the bandwidth available to a single connection (as will be described in Section 7.2). The techniques differ in their measurement methodology [144]: some employ direct measurement, others use projections (or mappings) from lower-level measurements, etc. Common to all the techniques is that they must first collect measurement samples (that are useful for the particular technique) and then apply a process we call *bandwidth modelling* to produce (a sequence of) bandwidth samples that characterize the bandwidth currently available to an application.

Bandwidth prediction. Bandwidth modelling unifies the various measurement techniques as it requires the techniques to produce a sequence of bandwidth samples. This sequence may then be resampled to form a periodic time series that can be analyzed by standard methodology [21, 43]. Based on a history of such a time series, a *bandwidth predictor* may be able to estimate the bandwidth available in the (near) future. Predictability of future resource availability depends on the stability and persistence of the network traffic (as defined in [10]), that is, on statistical properties of the aggregate network traffic.

Our focus lies on the first aspect, bandwidth modelling, as it forms the basis for prediction.

7.1.4 Information collection

There exist three basic approaches to collect information that can be used to model the bandwidth that is currently available to an application. These approaches can be distinguished according to the layer in the ISO/OSI protocol stack that they use to acquire information:

Application-level. Obviously, the application is in a good position to monitor the bandwidth it gets. Monitoring can be done either at the sender or at the receiver: e.g., a sender can monitor how fast it can pump data into the network[2], and a receiver can monitor at which rate the data is delivered by the network.

Transport-level. Since congestion-aware transport protocols gather a number of performance metrics to adapt the transmission rate of a connection such that it matches the current congestion state in the network, they have most of the information needed by a network-aware application readily available. E.g., in the case of TCP [176] parameters such as

[2]With a congestion-aware transport protocol, this rate is constrained by flow and congestion control.

the congestion window (*cwnd*), and round-trip time (*rtt*) are readily available. Other metrics such as loss rate, number and duration of timeouts, etc. can easily be determined. A number of models have been proposed that use such transport-level information to characterize the bandwidth of a single TCP connection, e.g., [108, 136, 27].

Network-level. While application-level and transport-level mechanisms can quite naturally monitor the performance of a single connection, their knowledge about network status, that is, about the aggregate traffic of competing flows, is rather limited. In contrast, if traffic is monitored at the network level, information about aggregate network traffic can be obtained. Knowledge about the behavior of aggregate network traffic can be useful in predicting the application-level sharing of bandwidth. Network-level performance metrics can be gathered, by querying routers [121] for instance.

7.2 Related work

The previous sections roughly classified the issues related to dynamic bandwidth estimation along several dimensions: discovery vs. monitoring, modelling vs. prediction, and the methods for information collection. This section reviews related work according to this classification. Section 7.2.1 summarizes the multitude of probing mechanisms developed for bandwidth discovery. Section 7.2.2 discusses the somewhat contradicting results as far as predictability of network traffic is concerned. The section also describes systems built for bandwidth prediction. Section 7.2.3 reviews the network-aware applications cited in Section 2.3 with regard to bandwidth monitoring. Sections 7.2.4–7.2.6 then cover related research on bandwidth monitoring according to the method used for information collection.

Although bandwidth discovery and prediction are not the focus of our work on dynamic bandwidth estimation, we give a brief account of related work in these fields. The issues of bandwidth discovery and prediction are orthogonal to the monitoring (bandwidth modelling) techniques investigated in this dissertation and would therefore be a useful addition to our monitoring system that will be described in Chapter 9.

7.2.1 Bandwidth discovery

We review probing techniques proposed in the literature that can be used to discover two important properties of a network path: bottleneck bandwidth and available bandwidth. Considering the vast body of research and the large number of probing tools [25], our account of related work is by no means complete—we only try to cover the major conceptual contributions to this field.

Bottleneck bandwidth. The packet pair technique described by Keshav [92] has been adopted in the *bprobe* tool by Carter et al. [29]. The fundamental idea behind the packet-pair approach (and the techniques sketched here, which are all derived from packet-pair) is that two packets, injected into the network back-to-back, will be spread out in time when they arrive at the bottleneck by the transmission delay of the first packet across the bottleneck. From this spacing (and

the size of the first packet) one can infer the bottleneck bandwidth[3]. Paxson [143, 141] identifies a number of limitations of packet-pair-based approaches (out-of-order delivery, limited clock resolution, bottleneck bandwidth changes, and multipathing), and devises a more accurate and more robust bottleneck estimation technique called *packet bunch modes*. Lai et al. [94] extend Paxson's work and implement the *nettimer* tool that allows receiver-based bandwidth estimation. In contrast to the end-to-end techniques mentioned so far, *pathchar* developed by Jacobson [80] tries to estimate peak-bandwidth on a hop-by-hop basis. Although *pathchar* can reveal valuable information about the characteristics of a particular network path, it can introduce quite significant network overheads that exceed those introduced by the other bottleneck bandwidth probing tools [44].

Available bandwidth. A straightforward method to measure available bandwidth (by means of probing) is to transfer considerable amounts of data across the network and measure the transfer time. There are at least three concerns with such a straightforward method: first, it must be ensured that the packets are injected fast enough into the network, that is, at a rate that exceeds the bottleneck bandwidth. The *cprobe* tool developed by Carter et al. [29] uses an estimate for the bottleneck bandwidth (obtained from *bprobe*) to set the transmission rate of the probing packets.

Second, probing must not be unfair to other competing traffic, that is, the data stream produced by probing tools must obey the Internet traffic rules defined by the well-established congestion control principles [79]. TReno [104, 101] is such a tool that mimics TCP's congestion control procedures while probing a particular network path for the bandwidth available. Care must be applied to send enough data so that the TCP connection reaches equilibrium, that is, the congestion avoidance phase [79][4]. Otherwise, slow-start may dominate the transfer, and the throughput experienced would not reflect the bulk transfer capacity actually available.

Third, it would be desirable if available bandwidth could be assessed without fully stressing the network path. Paxson [141, 143] sketches a technique that infers the available bandwidth from fine-grained measurements of the one-way transit times of data packets. More precisely, Paxson uses the variability in the one-way transit times to infer how much of the transit time of a particular packet can be attributed to queuing at the bottleneck router. Knowledge of the bottleneck bandwidth then allows to infer how much of the queuing delay experienced by a packet is due to the network load of its own connection, and how much of the queuing delay can be attributed to the network load of other connections. The ratio of these two constituents of the queuing delay experienced by a packet is proportional to the bandwidth available to this packet's connection. This technique has the advantage of getting estimates about available bandwidth without fully stressing the network. The major drawback is the difficulty of accurately inferring one-way transit times of packets [141, 142].

[3] Originally, Keshav [92] was interested in estimating *available* rather than *bottleneck* bandwidth by means of packet-pair probing. However, he had to assume that routers obey the "fair queuing" scheduling discipline, which is not presently the case in the Internet.

[4] For a connection with a bandwidth-delay product of $cwnd_{equilibrium} = bw \cdot rtt$ and zero losses, approximately $3 \cdot cwnd_{equilibrium}$ bytes must be sent in slow-start before the connection reaches equilibrium [27].

7.2.2 Bandwidth prediction

We can distinguish two areas of work in the context of bandwidth prediction. First, there are studies that analyze network traffic to gain insight about the predictability of properties of individual network paths. These studies address the question whether the network is "stable" enough to allow for network properties such as the bandwidth available to a connection to be predicted (e.g., by standard methodology [21]). A second group of research focuses on actually building such prediction services that can be asked to predict the properties of a particular network path.

Predictability. Analyzing the predictability of network-path properties (as seen by individual connections) has been the subject of two measurement studies [141, 10]. The first study by Paxson [141] investigates the evolution of loss rates. This aspect is important as it indirectly effects the throughput of congestion-aware protocols (see Section 7.2.5). The study discovers that the probability of seeing a zero-loss connection after having witnessed a zero-loss connection is approximately 0.75; this probability is fairly stable over time-scales of $10^2 - 10^4$ seconds. The same applies for connections that experience loss: such connections are again a good predictor for seeing a successive connection with non-zero loss. Furthermore, Paxson shows that the available bandwidth is also a good predictor for similar time scales.

The second study by Balakrishnan et al. [10] identifies two kinds of temporal evolution paths of the bandwidth for a particular sender-receiver pair: stable and unstable. (The authors do not report the fractions of all the connections they observed that fall into each of the two classes.) For the class of host-pairs with stable bandwidth, the authors note that bandwidth is within a factor of 2, even for time intervals of up to tens of minutes ($10^3 - 10^4$ seconds).

The time-scales investigated by these studies may be too large for some network-aware applications. In fact, a recent study by Veres et al. [186] seems to suggest quite the contrary to the observations above. Veres et al. state that TCP congestion control can exhibit chaotic behavior. The authors show that, contrary to common belief, TCP itself as a deterministic process can create chaos which generates self-similarity. (Self-similarity is a phenomenon that has been attributed mainly to *aggregate* TCP/IP traffic [96, 197] so far.) Veres et al. demonstrate the major features of chaotic systems in TCP/IP networks: unpredictability, extreme sensitivity to initial conditions, and odd periodicity.

In summary, these studies allow the following two contradicting conclusions to be drawn. On one hand, the findings by Balakrishnan et al. and Paxson seem to indicate that available bandwidth is fairly stable, thus, encouraging the use of bandwidth caches (or bandwidth prediction systems, see below) that may help clients select an appropriate mirror server, or may help alleviate start-up problems of network-aware applications (see Section 4.7). On the other hand, the empirical evidence presented by Veres et al. and also Balakrishnan et al. suggests that available bandwidth can fluctuate considerably enough to be termed unstable and unpredictable. Because it is currently not known which fraction of today's Internet traffic shows unpredictable behavior, the fact that unstable bandwidth can be observed implies that bandwidth must continually be monitored so that applications can adapt to bandwidth fluctuations to ensure predictable application behavior.

Prediction service. Based on the observations made by Balakrishnan et al. [10], Stemm et al. [163, 173, 174] developed SPAND (Shared PAssive Network performance Discovery), a system that collects bandwidth samples by passive monitoring on a shared network at a client site, caches the samples and predicts available bandwidth to a remote site based on the cached information. SPAND is used to aid clients in server selection [174].

Other research [145, 62, 182, 84] recognizes the increasing need for networked applications to be able to quickly and efficiently learn the distance, in terms of metrics such as latency or bandwidth, between Internet hosts (e.g., for server selection, network-aware delivery, etc.). If probing techniques were ubiquitously used by applications, the Internet could suffer severe scalability problems. To solve this problem of *scale*, Paxson et al. [145] and Francis et al. [62] propose an architecture, called IDMaps, for a *global* Internet host distance estimation service. (This proposal is in contrast to SPAND, which provides a local distance estimation service.) The architecture of IDMaps consists of a network of instrumentation boxes, called tracers, distributed across the Internet. These tracers periodically measure distances (e.g., in terms of bandwidth) among themselves to generate a distance map of the Internet. This service can then be queried by applications to obtain bandwidth estimates for an end-to-end path of interest. Obviously, not all possible paths on the Internet can be probed. However, if the tracers are reasonably placed [84, 182], metrics for other paths can be extrapolated from the measurements [62, 159].

Yet other work contributes more fundamentally to the construction of on-line prediction systems that could be used in systems such as SPAND and IDMaps, or in on-line monitors (see below). Dinda et al. [43] present RPS, an extensible toolkit for building flexible on-line and off-line Resource Prediction Systems in which resources are represented by independent, periodically sampled, scalar-valued measurements streams. RPS can be used to evaluate predictive models (that is, to perform predictability analyses) as well as to build on-line prediction systems, e.g, for host load or network bandwidth.

7.2.3 Bandwidth monitoring

Reviewing the related work on adaptive applications cited in Section 2.3 we note that only a few systems adapt to bandwidth fluctuations and therefore need information from a monitor. Furthermore, we note that most of these applications employ performance- or feature-based adaptation [171] and monitor loss rates (e.g., the video tools based on RLM [113], MTP [72], or RAP [153]) or buffer occupancy (e.g., HIPPARCH [93]) instead of available bandwidth. TOMTEN [39] relies on the user to detect changes in quality and employs bandwidth discovery techniques upon user intervention to learn about network status.

Of the related work cited in Section 2.3 only Odyssey [130] and Prayer [16] dynamically track bandwidth. These systems perform bandwidth monitoring at the application-level. Han et al. [70] performed some limited application-level monitoring at their transcoding proxy to collect measurement data that helped them to infer their static adaptation policies (Section 7.2.4).

Although there is limited work in bandwidth monitoring in the context of adaptive applications, a significant amount of research has been under-taken by the network community to better understand the dynamics of today's network traffic. Some of the insights gained have been applied to build an infrastructure for network performance monitoring, other results still await

"integration into applications". The following three sections review related work on bandwidth monitoring and on related problems with potential contributions to bandwidth monitoring.

7.2.4 Application-level monitoring

Many multimedia applications (e.g., vic [112], vat [81], or ivs [183]) use the two application protocols RTP and RTCP [162] to transfer their data and to advertise performance reports. RTP provides end-to-end network transport functions suitable for the transmission of real-time data over multicast or unicast network services. The control protocol (RTCP) enables monitoring of the data delivery in a manner scalable to large (multicast) networks. An application can infer metrics such as bandwidth, packet loss rates, round-trip times, etc. from the performance reports exchanged by means of RTCP. Bandwidth can be determined from the timestamps and the cumulative byte counts reported in subsequent performance advertisements. A key feature of RTCP is that it controls the frequency of these performance reports so that the amount of bandwidth consumed by the reports does not exceed a pre-defined fraction of the session bandwidth. (This feature is especially important for multicast sessions.) RTP and RTCP are independent of the underlying transport and network layers.

The Odyssey system [130] takes a slightly different approach: the central component of Odyssey is the so-called viceroy, which acts as the single point of control, and which is responsible for bandwidth estimation. For each application connection the viceroy maintains two observation logs: one for remote procedure calls (to estimate round-trip time) and one for bulk transfers (to estimate throughput)[5]. The viceroy periodically examines the recent transmission logs of all active connections and determines the instantaneous bandwidth available to the entire machine. It then estimates how much of that bandwidth is likely to be available to each connection in the coming period and notifies (i.e., up-calls) the application if the bandwidth strays out of the tolerance interval specified by the application. Odyssey's approach allows for a sharing of bandwidth information between different applications. However, this sharing is limited to the scope of the local area network.

Han et al. [70] take a similar approach for the evaluation of their transcoding proxy. They use an application-level connection monitor, which is implemented as a transparent shim layer (a winsock style dll on a windows platform) between the application and the socket layer. The connection monitor records every send and receive event at the proxy. Whenever an application (e.g., the proxy server) makes a socket layer call, a stub routine inside the shim layer is executed, which, after creating a log entry, passes control to the requested function inside the socket layer. The connection traces produced by the stub routines are then analyzed (off-line) by a statistical analyzer. With their approach, Han et al. are able to monitor both the server-proxy connection (with a receiver-based approach) and the proxy-client connection (with a sender-based approach). The differences of a sender- and a receiver-based approach to application-level monitoring are detailed in Section 7.3.2.

[5]These logs are produced by RPC2 [157], a user-level remote procedure call package that supports bulk transfers. RPC2 is used by all Odyssey applications [130].

7.2.5 Transport-level monitoring

Recently, considerable research efforts have been devoted to develop models that characterize the throughput of a single TCP connection. Two lines of research in TCP modelling can be identified.

First, a number of researchers devised and refined models for TCP's *steady-state* throughput. The term steady-state characterizes the behavior of long-running TCP connections in equilibrium. Thus, steady-state throughput is (by definition) a good measure for the bulk transfer capacity—the metric of interest to a network-aware application. Floyd [52] heuristically derives a simple analytical model for the throughput of a single TCP connection and finds that TCP throughput is proportional to $1/(rtt \cdot \sqrt{p})$, where rtt is the round-trip time and p is the probability of packet loss. Floyd's work is extended by Ott et al. [134] and Mathis et al. [108]. These two studies model the stationary (i.e., steady-state) behavior of ideal TCP congestion avoidance. The congestion avoidance mechanisms modelled are idealized in the sense that loss (of multiple) packets does not lead to timeouts. Mathis et al. empirically validate their models with Internet measurements. Independent work by Lakshman et al. [95] confirms these models. Padhye et al. [136] address the limitations of the previous models and incorporate the effects of TCP's timeout mechanisms on throughput. Their model also accounts for receiver-window limitations, that is, the model distinguishes between connections that are constrained by the network (congestion control) and connections that are constrained by a slow receiver (flow control).

Second, two studies propose models for the behavior of short TCP connections which are dominated by effects of the initial slow-start: to study the interaction of HTTP with different transport protocols Heidemann et al. [71] develop a performance model for TCP throughput in slow-start under the assumption of zero packet loss. Cardwell et al. [27] extend the work of Heidemann et al. and Padhye et al. [136] and present a unified model for TCP performance for both short and long TCP connections and a wide range of loss rates.

There are several usage scenarios for TCP throughput models: First, these models have been developed for conformance testing of other congestion-aware transport protocols (e.g., reliable multicast protocols [187], or multimedia streaming protocols [153]). A protocol is termed conformant if it exhibits the behavior of TCP-friendly traffic flows [103]. The TCP models are used to accurately characterize what is meant by TCP friendliness. Second, these models can be used for network provisioning and bandwidth allocation in the Internet [108]. Third, the throughput models can be used for the simulation of large-scale TCP/IP networks [75]. Fourth, and most importantly for our concerns, these models can be used for transport-level monitoring to estimate available bandwidth for network-aware applications [18]. The main feature that makes these models attractive from an application's point of view is that they focus on the important factors (loss rate, round-trip time, etc.) that effect the long-term behavior of a TCP connection and are therefore well suited to *predict* future performance. However, to date, no bandwidth monitor exists that uses such models.

7.2.6 Network-level monitoring

One of the drawbacks of bandwidth estimation based on application-level and transport-level monitoring is that these approaches have only limited knowledge about competing traffic. In

contrast, querying the network itself allows an exact picture of the status of the entire network to be obtained [100]. The network topology revealed this way allows to predict application-level sharing. Furthermore, the cost (in terms of network load) of network-based queries is linear in the size of the network. (This property is in contrast to benchmark-based or application-level measurements whose costs are often quadratic in the number of network nodes.) The Remos system [99, 121] developed at CMU queries routers by means of SNMP [170] to obtain network status information. The status information revealed by such queries can be represented by an annotated graph whose edges are the links traversed by a connection [99]. The annotations reflect the performance metrics of interest, e.g., the link bandwidth and the fraction of the link bandwidth used by the aggregate traffic at that router. The key challenge with such an approach is the development of a mapping function which maps network status, that is, the list of link bandwidths, to the bandwidth available to an application on an end-to-end basis [100]. The Remos implementation is a two-tier approach consisting of collectors and modellers [121]. Collectors are responsible for network-oriented functionality and collect static and dynamic network information that is relevant to applications. Remos implements three types of collectors [121]: SNMP-collectors, benchmark-collectors (for parts of the network whose routers do not respond to SNMP queries; benchmark collectors use probing techniques such as described in Section 7.2.1), and master collectors (to organize collectors in a hierarchy). A modeller is responsible for application-oriented functionality, e.g., to map the network-level metrics to performance information that is meaningful to the application.

7.3 Qualitative comparison

This section gives a qualitative comparison of the related work on bandwidth monitoring presented in the previous sections. As our focus lies on bandwidth monitoring (and bandwidth modelling), we do not treat issues related to bandwidth discovery and prediction here. One of the questions we want to address in the remainder of this dissertation is: "which of these approaches to bandwidth monitoring is best suited to fulfill the information needs of the network-aware applications studied?" To address this question, the following section lists the requirements for a bandwidth estimator and identifies the criteria for the comparison.

7.3.1 Criteria

With our sender-initiated approach to network-aware delivery (cf. Chapter 3), bandwidth information must be made available to the *sender*. This is in contrast to other systems, where bandwidth information is made available to the receiver, e.g., for the purpose of client-initiated adaptation [130] or server selection [28, 174]. In addition, there are two types of quantitative requirements for dynamic bandwidth estimation: the *quality* of the bandwidth estimates and the *efficiency* with which the estimates can be obtained by an interested application. Furthermore, there is the (recurring) requirement that solutions should be reasonably easy to deploy.

Quality. Based on the observations made in Section 6.7 there are two dimensions that define the quality of bandwidth estimates: first, the bandwidth estimates provided to the application must *accurately* reflect the bandwidth (that will be) available to an application. Second,

these estimates must be provided in a *timely* fashion, that is, a bandwidth monitor must detect and report changes in available bandwidth quickly.

As far as *accuracy* is concerned we distinguish three subcriteria: first, how well can the performance measures collected be mapped to the bandwidth available to a particular application? In other words, how well does this mapped (or modelled) bandwidth match the application behavior? Second, how well can the factors that effect available bandwidth be differentiated (e.g., is a particular monitoring approach suited to distinguish between host- and network-related effects on performance)? And third, can the effects of multiple applications sharing the network on the bandwidth available to applications be predicted?

Efficiency. Bandwidth estimates should be obtainable at small costs. There are two efficiency concerns: first, monitoring should impose as little overheads on the network as possible. Ideally, monitoring should not incur any network overheads at all. Second, the processing overheads should not impact the performance of the end-systems. In other words, monitoring should be scalable so that a large number of simultaneous connections can be handled without incurring significant overheads that distort network and application performance.

Deployment. Two aspects of interest can be subsumed under the term "deployment": First, does the monitoring approach require cooperation with other entities (end-systems, or routers) in the network? And second, which parts of a networked application have to be changed to implement a particular monitoring approach? Ideally, a monitoring approach requires as little cooperation with other entities as possible (both to ease deployment and to reduce network overhead). Furthermore, changes to existing systems or applications should be kept to a minimum.

7.3.2 Alternatives

Instead of comparing all the individual systems or approaches proposed in the literature, we categorize and discuss the monitoring approaches on a more abstract level. We compare four alternatives:

Application-level/Sender-based. The sending application monitors how fast it can pump data into the network. Because the data is typically buffered (by the socket layer), the application actually monitors the rate with which it can place data in the socket buffer. Once the send buffer is full, the rate with which the application can place new data in the buffer is constrained by how fast the network and/or the receiver can process data, that is, the rate is constrained by congestion and/or flow control. (Such a sender-based approach has been taken by Han et al. [70] to track the bandwidth of proxy-client connections.)

Application-level/Receiver-based. The receiving application monitors how fast the data is delivered by the network. The receiver periodically informs the sender about its status by reporting the number of bytes received since the last performance report. (Many of the applications using RTP/RTCP [162] pursue this approach. Similarly, Han et al. [70] monitor the server-proxy

connection at the proxy (for off-line analysis). Odyssey [130] also uses this approach, however, no performance reports have to be sent to the peer, as adaptation is receiver-initiated.)

Transport-level. A transport-level monitor consists of a model that characterizes application throughput (e.g., one of the models described in Section 7.2.5) and a method to collect the parameters (loss rate, round-trip time, etc.) that are relevant to this model. There are two options for information collection (e.g., for TCP): first, a packet sniffer such as *tcpdump* [111] can trace network traffic of individual protocol connections. The packet traces can then by analyzed by a tool such as *tcpanaly* [140, 141] that "reverse-engineers" the connection state (i.e., congestion window, round-trip time, loss rate etc.). Second, the data can be obtained directly from the sender's TCP stack (see Chapter 9). Because the relevant parameters are used by the transport protocol's congestion control procedures and because these procedures are typically employed by the sender to control the transmission rate, a transport-level monitor must be deployed at the sender.

Network-level. Network-level monitoring repeatedly queries the status of all the routers on the network path(s) taken by the connection(s) of interest. The router status includes information about link bandwidths and bandwidth consumed by the aggregate traffic at the router. The status (or performance reports) of all the routers of a network path are then combined to model the bandwidth available to an individual application. (This is the approach taken by the Remos system [99, 100].)

7.3.3 Comparison

Table 7.1 summarizes the qualitative comparison of the four approaches given in this section. We cannot comment on the issue of timeliness of the bandwidth estimates as timeliness mainly depends on the sampling frequency and the averaging interval used. Both parameters must be set by an application using a bandwidth monitor. Note, however, that the costs of collecting bandwidth samples may place an upper bound on the sampling frequency, and that these upper bounds may differ for the various monitoring techniques.

Application-level. The main advantage of an application-level approach is that it monitors the bandwidth the application actually experiences. Second, application-level monitoring is simple to implement (as demonstrated by previous work), and the end-system overheads incurred by monitoring a single connection are small. Furthermore, only the application must be changed to implement application-level monitoring (e.g., send and receive operations must be dispatched to a stub layer [70, 130]).

On the down side we note that it may be difficult for the application to differentiate between the various factors that effect performance. For instance, an application-level monitor cannot find out whether the bandwidth of a connection is limited by the sender, the receiver (flow control), or the network (congestion control). Distinguishing the effects of the various factors is useful because it allows to estimate available bandwidth in a more robust manner. E.g., transient behavior (slow-start, timeouts, etc.) can be identified and correctly be taken into account for the estimation. Moreover, an application-level approach cannot predict the effects of sharing

		Application-level Sender	Application-level Receiver	Transport-level (Sender)	Network-level
Quality	Match application	± good	+ excellent	± good	− difficult
	Differentiate factors	− difficult		+ simple	± possible
	Predict sharing	− limited to local network			+ possible
	Timeliness	depends on sampling frequency and averaging interval			
Efficiency	Network overhead	+ none	− feedback	+ none	± queries
	End-system overhead	+ small	± small	+ small[a]	− medium
Deployment	Cooperation	+ none	− receiver	+ none	− router
	Changes	+ sender	± both peers	− kernel[a]	+ sender

[a]Tradeoff between efficiency (in terms of end-system resource usage) and ease of deployment (see Section 7.3.3).

Table 7.1: Qualitative comparison of monitoring approaches. (The symbol ± indicates that the particular factor is considered to be a positive feature of the particular approach, but, there are other approaches that compare (slightly) more favorably with respect to this factor.)

the bandwidth with competing traffic. This effect on application performance is impossible to predict by application-level monitoring beyond the limited scope of the local network.

Furthermore, we expect that bandwidth estimates obtained by a receiver-based approach more accurately match the bandwidth experienced by the application than those obtained by a sender-based approach. Sender-based monitoring only monitors the rate with which the new data can be placed in the send buffers. Even though this rate is often dictated by congestion control (and hence the network), we expect that bandwidth estimates derived from sender-based measurements exhibit significant transients (see Chapter 9). On the other hand, receiver-based monitoring has the disadvantage that it incurs network overhead for the transmission of the performance reports and that it requires cooperation of both peers. The network overhead may not be dramatic for a single application, but it may lead to scalability problems on a larger scale. Consider a network-aware server that maintains a large number of client connections that must be monitored. The feedback messages from these clients may lead to a phenomenon known from reliable multicast as feedback-implosion, which has a negative impact both on network and end-system performance. A consequence of this scalability problem is that the timeliness of receiver-feedback may suffer.

Transport-level. The advantage of the transport-level approach is that bandwidth available to an application can be matched quite accurately (if the appropriate models are chosen [18]; Chapter 8), even though the monitor must model (and hence approximate) application throughput. In addition, network and end-system related performance limitations can easily be differentiated. Furthermore, no network overheads are incurred, the end-system overheads are small and scale better with large numbers of connections compared to receiver-based, application-level monitoring (see Chapter 9). Note, however, that there is a price (in terms of deployability) to pay for efficiency: an efficient approach requires (small) modifications to the protocol stack (see Chapter 9). Otherwise, that is, if transport-level monitoring is done at the user-level (by means

of *tcpdump* [111] and *tcpanaly* [140]), considerable end-system overheads must be witnessed[6]. Another drawback is the inability to predict sharing effects beyond a limited scope.

Network-level. The biggest benefits of network-level monitoring (e.g., when compared to an application-level, receiver-based solution) are the wealth of valuable performance information revealed, and the scalability to large numbers of network-aware applications (that is, monitored connections). The second benefit stems from the fact that a large number of connections may have a significant portion of traversed network nodes in common. Such a situation allows to share status information between the monitored connections. As a consequence, available bandwidth may be predicted more accurately, because bandwidth sharing effects can be taken into account. Furthermore, the network as a whole can benefit from considerable savings in network overhead.

The flip side of a network-level approach from the perspective of an individual application is that it may be difficult to accurately map network-level performance information to the bandwidth available to the application. (In fact, for their experiments with network-level bandwidth prediction, Lowekamp et al. [100] had to calibrate their network-to-application mapping with measurements from an application-level monitor.) Although a network-level solution may allow for a sharing of information (and thus reduce the overhead for the network as a whole), probing each router on a particular network path incurs overheads that can be significant from the point of an individual application (e.g., the end-system must process the responses to these queries). Moreover, not all routers along a network path of interest may be cooperative and respond to queries. Such a situation may compromise the usefulness of the network-level approach as it calls for application-level benchmarking to learn about network performance in those networks (this is the reason for the existence of benchmark collectors in Remos [121]).

7.3.4 Discussion

Network-level resource monitoring can provide a wealth of detailed information about network status (topology, link bandwidths, bandwidth utilization of links, etc.). However, it is ill-suited for the type of network-aware applications studied in this dissertation for two reasons: first, we expect the costs of obtaining such detailed status information to be fairly high (compared to the costs incurred by other approaches). Second, the network-aware applications studied here are mainly interested in end-to-end performance. However, such information is non-trivial to obtain by means of network-to-application mapping. Thus, the costs of topology discovery and querying routers are not justified for this type of application (except for potential benefits due to "sharing effects" among multiple applications). Other applications, e.g., parallel applications that involve multiple hosts to carry out a complex computation [67, 171], or multi-party applications that use intermediary nodes in the network to perform bandwidth adaptation for different receivers [72, 90], may be better served by a Remos-type network monitor.

The comparison of the remaining three approaches does not yield a conclusive answer to the question "which of the approaches to bandwidth monitoring is best suited to fulfill the information needs of the network-aware applications studied?". Whether application-level or

[6] Paxson [140] notes that the development of *tcpanaly* was not fully satisfying because of the inability to write the tool in terms of a one-pass analysis for generic TCP actions.

transport-level monitoring is the better solution for a network-aware application such as Chariot depends on a number of issues. E.g., how do the two approaches compare (quantitatively) with respect to accuracy and overheads? How important is the ability to differentiate between the various factors effecting the bandwidth available to an application? How accurately can the available bandwidth be characterized by the transport-level models described in Section 7.2.5? How big are the changes to the protocol stack to enable transport-level monitoring? We address these questions as follows:

First, in Chapter 8 we present an in-depth evaluation of different models that can be used for transport-level monitoring as a result of an in-vivo experiment in the Internet. The experiment allows us to draw conclusions with respect to the accuracy of TCP models. In addition, the experiment allows us to gain some insight on the following two questions[7]: how does available bandwidth (defined as the bulk data transfer capacity [102]) depend on the transport protocol used to transfer the data? Is there empirical evidence that would support or refute any of the contradicting reports on the predictability of available bandwidth as experienced by an individual connection?

Second, to quantitatively assess and compare the efficiency and accuracy of different monitoring approaches, we must build a flexible infrastructure for bandwidth monitoring that allows to collect bandwidth information by means of different techniques simultaneously. Chapter 9 presents the architecture and implementation of our monitoring toolkit and provides a comparative evaluation of application-level and transport-level monitoring.

7.4 Summary

Since the performance of network-aware applications by and large depends on the timeliness and accuracy of the bandwidth estimates, the second part of the dissertation (Chapters 7–9) discusses different approaches to obtain information about network status. This chapter recapitulates the requirements for dynamic network resource estimators: bandwidth monitoring (and this includes modeling of available bandwidth) and prediction of bandwidth availability.

The two issues are orthogonal. Our focus lies on the first aspect: bandwidth monitoring. The chapter classifies and discusses different approaches to obtaining bandwidth estimates (application-level monitoring, transport-protocol cooperation, and direct queries of routers). The review and qualitative comparison of related work on bandwidth monitoring allows us to draw the conclusion that network-level monitoring is ill-suited to support the type of network-aware applications considered in this dissertation. Furthermore, the chapter identifies that more work is needed to understand the implications and potential benefits of transport-level monitoring, and it identifies the need for a flexible architecture of a monitoring toolkit that enables a quantitative comparison (in terms of efficiency and accuracy) between application-level and transport-level approaches to bandwidth monitoring.

[7]These questions have been identified to have been left unanswered by previous research (see Section 7.1.1 and 7.2.2 respectively).

Seite Leer / Blank leaf

Chapter 8

Transport-level bandwidth modelling

The comparison of different monitoring approaches in Chapter 7 indicates that transport-level monitoring may be an attractive approach to dynamic bandwidth estimation (for reasons of efficiency and accuracy). By means of an in-vivo Internet experiment we want to address the following three questions identified in the previous chapter: First, how accurately can the bandwidth available to an application be modelled by transport-level (i.e., TCP-level) throughput models? Second, how does the choice of the transport protocol (i.e., TCP variant) effect the available bandwidth? Third, is there empirical evidence that would help to support (or refute) conjectures about the stability and predictability of available bandwidth put forth by related work.

Section 8.1 presents the TCP throughput models to be evaluated. Section 8.2 introduces the evaluation methodology. Sections 8.3 and 8.4 compare the accuracy of the TCP models studied and try to explain the differences between the models observed. Section 8.5 assesses the effect of different TCP variants on the accuracy of the models. Section 8.6 addresses the question related to the predictability of available bandwidth.

TCP terminology

The following sections assume that the reader is familiar with the basic concepts of TCP's congestion control mechanisms (see [79, 176, 4] for references). Here we briefly introduce a few terms and abbreviations that are frequently used in this chapter.

TCP treats packet loss as a signal for congestion. When a TCP sender notices the loss of a data packet it reduces the congestion window (denoted as *cwnd*). The congestion window constrains the amount of data that can be sent within one round-trip time (*rtt*). There are two alternatives how packet loss can be *indicated* to a TCP sender. The sender can assume that a packet has been lost either if it receives three duplicate acknowledgments, or if it does not receive any acknowledgments (acks) for a significant period of time (the so-called *timeout period*). In the first case, TCP reacts to the congestion indication by halving *cwnd*. In the second case, that is, when the retransmission timer expires, the *cwnd* is reduced to 1 packet. The two types of loss indications are termed *triple duplicate ack (TD)* and *timeout (TO)* loss indication, respectively.

8.1 Throughput models

We consider four simple models: one (M_1) is directly derived from the TCP specification, the other three (M_2, M_3, and M_4) have recently been described in the literature [108, 136, 27]. Let bw_i denote the bandwidth predicted by model M_i.

M_1 (TCP specification [4]). At any instant during a data transfer, TCP's measure of the available bandwidth is given by the size of the congestion window (*cwnd*) and the measured round-trip time (*rtt*) [79, 4]. Thus, a straightforward approach is to simply use the averages of *cwnd* and *rtt* to characterize the average application-level throughput.

For M_1 we compute the weighted average of the number of packets the connection kept in flight, that is, we compute $\overline{cwnd}$ as the weighted average of the *cwnd* size over the duration of the whole connection, where the weights are given by the time *cwnd* stays constant[1]. Then,

$$bw_1 \approx \frac{(\overline{cwnd})}{rtt},$$

where *rtt* is the mean of the round-trip time samples produced by the protocol.

This model is simple to compute. Exposing the parameters to an application would require only small interface changes. On the other hand, the model is applicable only to TCP and may be too simple to be accurate.

M_2 (Mathis et al. [108]). The throughput model for ideal TCP congestion avoidance developed by Mathis et al. computes bandwidth as a function of the loss indication probability p, the round-trip time *rtt*, and the packet size *mss*:

$$bw_2 \approx \frac{mss}{rtt}\frac{C}{\sqrt{p}}.$$

The constant C depends on the receiver's acknowledgment strategy and on assumptions about the occurrence of packet loss[2]. The congestion avoidance behavior modelled by M_2 is idealized in the sense that packet loss does not lead to timeout phenomena. This statement implies that the transport protocol is assumed to recover from multiple packet loss (in a single congestion window) purely with fast retransmit/fast recovery (which is triggered by the reception of three duplicate acknowledgments [4]). Thus, the loss indication probability p is defined as $\frac{TD}{dataSent}$, where *TD* is the number of triple duplicate acknowledgment events experienced. *dataSent* is the total number of packets sent (i.e., including retransmissions).

The advantage of this model is that it is mainly dependent on network-layer metrics, and hence also applicable for purposes other than modelling TCP throughput (see Section 7.2.5). Second, the model is simple to compute as it is based on a minimum of information. On the

[1] We account for the artificial inflation of *cwnd* in fast recovery [176, 4] by only using the constant number of packets in flight during recovery instead of the actual value of *cwnd*.

[2] For delayed acknowledgments [4], Mathis et al. [108] use $C = \sqrt{3/4} \approx 0.87$ under the assumption of periodic loss. Ott et al. [134] compute a value of $C \approx 0.93$ under the assumption of random loss. Padhye et al. [136] show that the constant C depends on the strategy with which the receiver acknowledges data packets. If the receiver acknowledges every b-th data packet then $C(b) = \sqrt{\frac{3}{2b}}$. Current TCP implementations typically use $b = 2$.

other hand, the model only captures asymptotic, that is, steady-state TCP congestion avoidance behavior (and thus assumes that packet loss *must* occur). Therefore, the model is not suited for short connections, and it is undefined for connections without loss. Furthermore, accuracy may be low because the model does not consider timeouts.

M_3 (Padhye et al. [136]). The model M_3 extends M_2 by taking into account the effects of retransmission timeouts (RTOs) and receiver-window limitations (i.e., flow control).

To model timeout effects, the expected number of timeouts in a connection and the average duration of a timeout period must be computed. For the first task, Padhye et al. express the probability that a particular packet loss event happens to be a timeout as a function $Q(p,b,w)$ of the loss indication probability p, the frequency b with which the receiver acks data packets, and the size of the congestion window w. Intuitively, the higher the packet loss probability the higher the probability that not enough acks arrive at the sender to trigger fast retransmit/fast recovery, and thus, the more likely it is that a timeout occurs. Similarly, the smaller the congestion window, the smaller the number of packets a connection can keep in flight, and the more likely it is that not enough duplicate acks arrive. For the second task, the estimation of the average duration of a timeout, Padhye et al. rely on the users (of their model) to specify the parameter t_{RTO}, the average duration of the first timeout in a sequence of one or more successive timeouts. Because timeouts are subject to exponential back-off (if a timeout retransmission is lost), the average duration of a timeout can be expressed as a function of t_{RTO} and the packet loss probability p.

The bandwidth of a TCP connection in steady-state can either be limited by congestion control, which is modelled as sketched above, or by flow control. A connection that is limited by flow control is called receiver-window constrained. The throughput of a connection constrained by a receiver window W_{max} can be approximated by $\frac{W_{max}}{rtt}$.

More specifically, Padhye et al. [136] develop M_3 as follows: they approximate the expected congestion window size $E[w]$ as $\sqrt{\frac{8}{3bp}}$. The loss indication probability p is defined as $\frac{TD+TO}{dataSent}$, where TO is the number of timeouts experienced by a connection. From $E[w]$ the throughput of TCP connections without timeouts is shown to be inversely proportional to $B(p,rtt,b) = rtt\sqrt{\frac{2pb}{3}}$.[3] The probability that a packet loss results in a timeout is given by the function $Q(p,b,w)$. Using $E[w]$, the expected number of timeouts can then be computed as $Q(p,b) \approx min(1, \frac{3}{E[w]})$. Taking into account that timeouts are subject to exponential back-off, the average duration of a timeout can be approximated as $t_{RTO}p(1+32p^2)$. The model proposed by Padhye et al. approximates bw_3 as a function of p, b, rtt, mss, t_{RTO}, and W_{max}:

$$bw_3 \approx min\left(\frac{W_{max}}{rtt}, \frac{mss}{B(p,rtt,b) + t_{RTO}\,Q(p,b)\,p(1+32p^2)}\right)$$

Since the model is based on a more comprehensive set of parameters, the accuracy should be high. On the other hand, although the model is still fairly simple, it needs more information (which is harder to get) than M_2. Like M_2, M_3 models asymptotic behavior of congestion avoidance. Therefore, it is not suited for short connections and is undefined for connections with zero loss.

[3] Note that $\frac{mss}{B(p,rtt,b)}$ is equivalent to bw_2, the throughput as modelled by M_2.

M_4 (Cardwell et al. [27]). The work by Cardwell et al. extends M_3 by taking slow-start behavior into account. The authors model TCP latency for a transfer of *dataSent* *mss*-sized data segments by decomposing the data transfer latency into three major aspects: the initial slow-start phase, the resulting packet loss (if any), and the transfer of the remaining data. Cardwell et al. first calculate the expected amount of data $E[d_{ss}]$ that can be transferred in the initial slow-start phase before encountering a packet loss or finishing the data transfer. Using $E[d_{ss}]$, the time spent in slow-start $E[T_{ss}]$, the final congestion window at the end of slow-start $E[W_{ss}]$, and thus the expected cost of loss recovery $E[T_{loss}]$, if any, can be deduced. Then, the steady-state throughput model developed by Padhye et al. [136] is used to approximate the cost $E[T_{ca}]$ of sending the remaining data. M_4 then models the expected throughput of a connection that transfers $dataSent$ packet as:

$$bw_4 \approx \frac{dataSent \cdot mss}{E[T_{ss}] + E[T_{loss}] + E[T_{ca}]}$$

To model $E[T_{ss}]$ two scenarios must be distinguished. First, if a connection experiences no packet loss ($p = 0$), then all packets can be sent in slow-start. However, due to limitations imposed by the receiver window or the default slow-start threshold [4], the exponential growth of the congestion window may not be unbounded. To reach a congestion window size of w, $E[d_{ss}] = \frac{\gamma \cdot w - w_1}{\gamma - 1}$ data packets must be sent, where w_1 is the initial congestion window size[4], and γ is the rate with which the sender opens the congestion window in each round-trip time. γ is a function of b, the receiver acking-strategy ($\gamma = 1 + 1/b$). Second, if a connection experiences packet loss ($p > 0$), the expected number of data packets that can be sent in slow-start before a loss occurs ($E[d_{ss}]$), can be expressed as a function of p. (The basic idea is that the number of packets that a sender can transmit without experiencing a loss follows a binomial distribution.)

For both cases, $E[T_{ss}]$, the time required to transmit $E[d_{ss}]$ data packets in slow-start, depends on γ and the initial size of the congestion window w_1. The number of round-trips r required to send $E[d_{ss}]$ is given as $log_\gamma(\frac{E[d_{ss}](\gamma-1)}{w_1} + 1)$. Then, we have $E[T_{ss}] = r \cdot rtt$.

$E[T_{loss}]$, the expected time needed to recover from the first loss, can be computed as follows. With the probability $Q(p, w, b)$, where $w = E[W_{ss}]$, the packet loss results in a timeout, in which case $E[T_{loss}] = t_{RTO}$; otherwise, the loss can be recovered within one *rtt* by using fast retransmit/fast recovery. Finally, the time $E[T_{ca}]$ required to transfer $dataSent - E[d_{ss}]$ packets in congestion avoidance is computed using the throughput approximation given by M_3.

We note that M_4 uses the same parameters as M_3, thus, similar arguments apply. In contrast to M_3, M_4 is applicable to all types of connections, including short connections and connections with zero loss.

8.2 An Internet experiment

To generate a collection of traces that we can use to validate (or refute) our conjectures about the different TCP throughput models, we performed an in-vivo experiment in the Internet. Here we briefly summarize the setup of the experiment, see [19] for details.

We produced a family of user-level protocol implementations to study the effectiveness of standard TCP Reno congestion control as well as to evaluate various enhancements to recover

[4]Typically, TCP implementations use $w_1 = 1$. Recently, the use of $w_1 = 2$ has been permitted [4].

from or to avoid congestion [19]: Vegas, SACK/FACK, Rate-Halving (RH), and Rate-Halving with Lost Retransmission Detection (LRD)[5]. We then installed a daemon that is based on Paxson's network probe daemon [141] at a number of Internet hosts in North America (CMU, MIT, UC Irvine, DEC SRC Palo Alto[6], U Waterloo Canada) and Europe (universities in Zurich, Berne, Linz, Munich, Madrid, and Turku). From time to time, the daemons at randomly selected source and sink hosts were contacted and instructed to carry out transfers of 1MB of data using a randomly chosen protocol variant. (To ensure that the receiver window does not limit the bandwidth available to a connection, we chose large windows of 256kB.) The intervals between successive transfers between a given pair of hosts are taken from an exponential distribution with a mean of 10 hours. We recorded detailed measurements of the micro-dynamics of each connection. These daemons were active over a 6-month period (from November 1997 till May 1998). The experiment includes the records of 24943 connections: 4148 Reno, 4100 Vegas, 5415 FACK, 5569 Rate-Halving (RH), and 5711 LRD connections.

8.3 Evaluation of models

The usefulness of throughput models for the needs of network-aware applications rests on the accuracy of the information: How well does a model capture the bandwidth currently available?

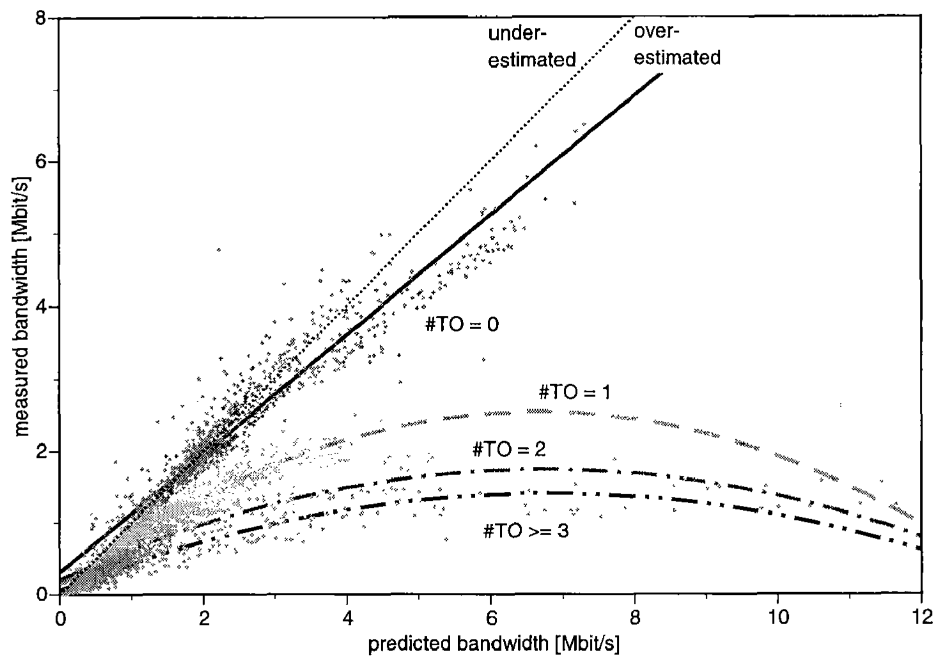
To assess the accuracy of a TCP throughput model M_i, we compute the throughput bw_i as predicted by the model for each of the connections of our experiment (for which the model is applicable) and compare bw_i with the throughput $bw_{measured}$ actually experienced by the application. We compute $bw_{measured}$ as $\frac{dataSent \cdot mss}{t_{transfer}}$, where $t_{transfer}$ is the duration of the 1MB data transfer measured. To compute bw_2, bw_3 and bw_4 we use the following (measured) loss indication probability $p = \frac{TD+TO}{dataSent}$. TD is the number of triple duplicate ack events, and TO is the number of timeouts experienced by the connection.

With these definitions, we can use a scatter plot of the pairs $< bw_i, bw_{measured} >$ to verify the accuracy of the model M_i by checking whether a clear linear relation is discernible or not. Furthermore, we attempt to statistically characterize the "goodness" of a model by computing the coefficient of determination R^2 for a simple least-squares fit of the data. (R^2 is the square of the sample correlation and denotes the fraction of the variation that is explained by the regression [82]).

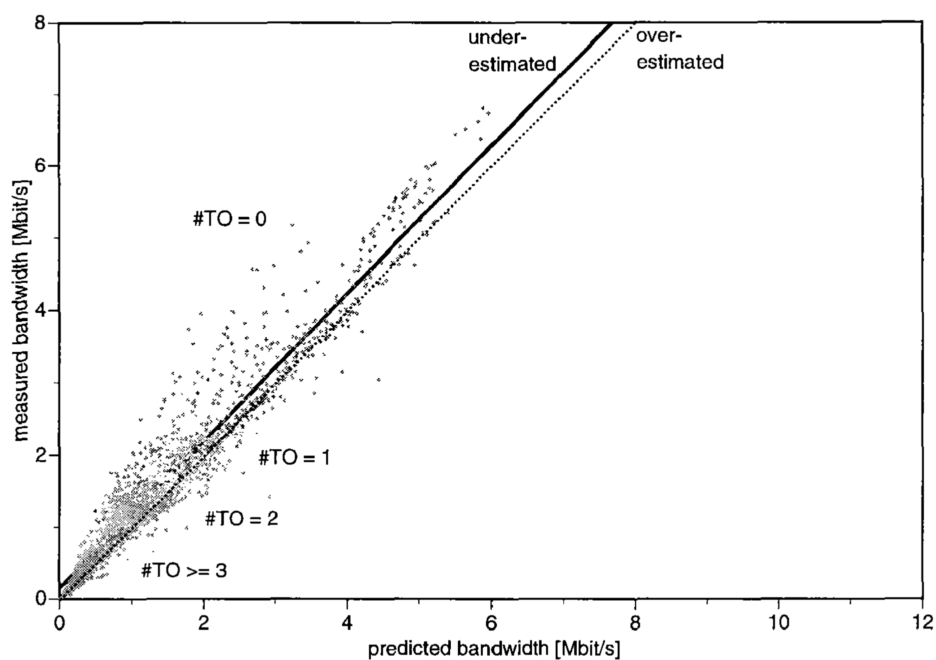
Figure 8.1 (a) depicts these pairs $< bw_1, bw_{measured} >$ for M_1. Clearly, M_1 is far from being close to an accurate model. In many cases, it grossly overestimates the bandwidth achieved. This deviation is not surprising since the $\overline{cwnd}$ also includes the idle periods before retransmission timeouts. We clustered the samples according to the number of timeouts experienced by a connection ($TO = 0, 1, 2$, and $TO \geq 3$). We tried to fit the clusters with quadratic functions simply to visualize the tendency of the overestimation to increase with TO. A similar tendency can be observed when looking at the data for M_2 (not shown); again this result is not surprising

[5]The protocol termed Vegas implements the novel congestion avoidance mechanisms proposed by Brakmo et al. [22]. The protocol variant termed FACK (forward acknowledgment) is based on ideas presented by Mathis et al. [105] that take advantage of the information conveyed by selective acknowledgments (SACKs) [107] to help improve TCP's congestion control. Rate-Halving [106] extends the mechanisms present in FACK. LRD [106] allows TCP to recover from lost (fast) retransmissions. These protocols are described and compared in [19].

[6]Now Compaq SRC Palo Alto.



(a) M_1



(b) M_4

Figure 8.1: Scatter plot for all Reno connections; clustered according to the number of timeouts per connection.

R^2		M_1	M_2	M_3	M_4
all connections		0.74	–	–	0.96
no loss	$(p=0)$	0.99	–	–	0.97
lossy	$(p>0)$	0.62	0.83	0.92	0.91
lossy, no timeouts	$(p>0 \wedge TO=0)$	0.99	0.85	0.85	0.86
lossy, timeouts	$(TO>0)$	0.63	0.86	0.93	0.93

Table 8.1: Summary of models (Reno connections).

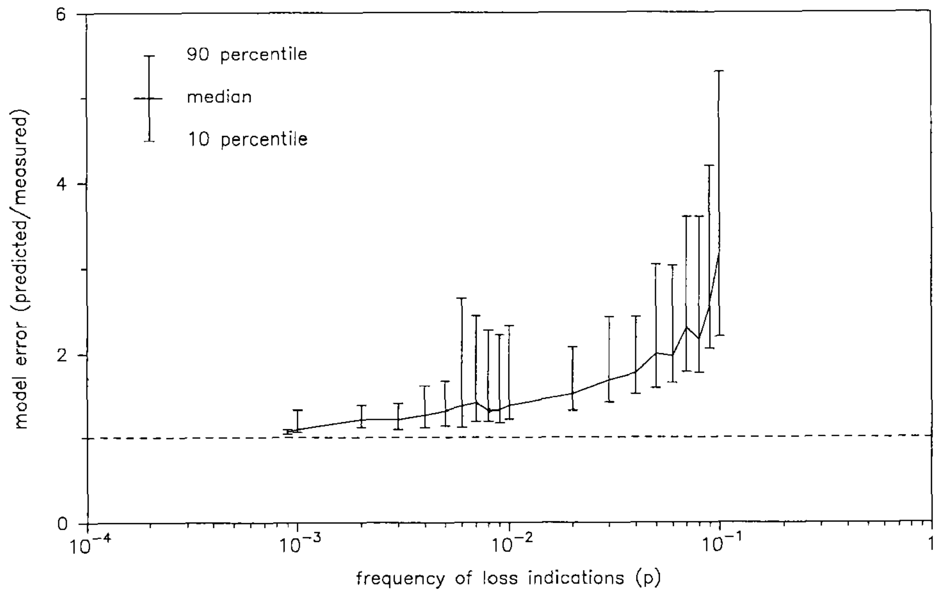
either since the model does not take retransmission timeouts into account [108]. In contrast to M_1 and M_2, which both overestimate the measured throughput considerably (in the presence of timeouts), M_3 and M_4 seem to model the measured data well, independent of the number of timeouts experienced. The data for M_4 is shown in Figure 8.1 (b), the results for M_3 (not shown) are almost identical. These findings are consistent with the results reported in [136] and [27].

These observations are also supported by the simple statistical analysis summarized in Table 8.1. This table shows a statistical comparison of the models for Reno connections based on different clustering criteria of the data. Note that M_2 and M_3 are not applicable to loss-free connections. The table allows us to comment on four aspects.

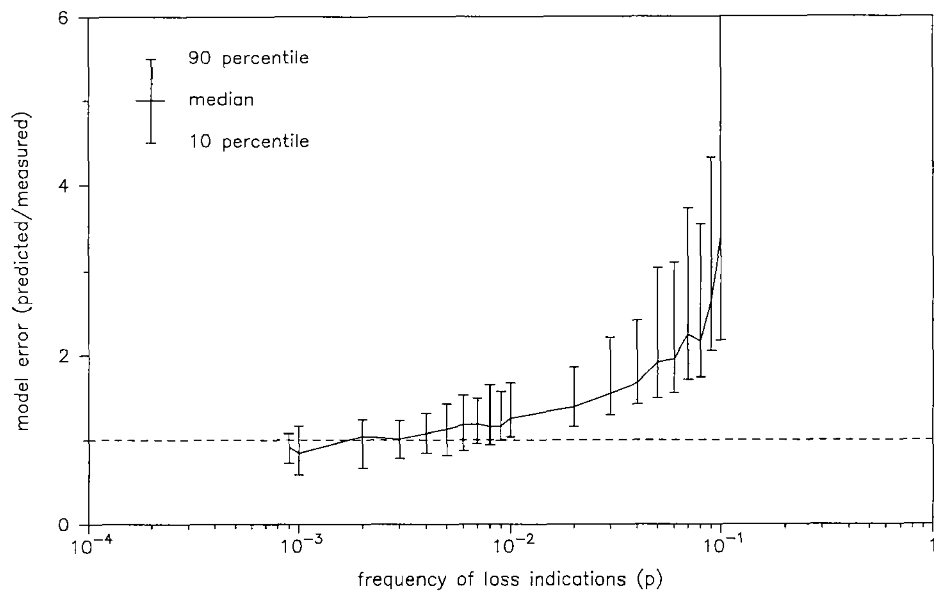
First, a comparison of the quality of the models for loss-afflicted connections shows that M_3 and M_4 best model the effective throughput, followed by M_2 and M_1. To gain further insight about the behavior of the models for different loss rates, we plot the relative model error, that is, the ratio of predicted and measured bandwidth ($\frac{bw_i}{bw_{measured}}$), for the range of loss indication probabilities p encountered. Figures 8.2 (a)–(d) show the graphs for M_1, M_2, M_3 and M_4 respectively. The x-axis represents the frequency of loss indications p on a log-scale. The y-axis shows the median and the 10 and 90 percentiles of the relative model error. Note that y values of 2 and 0.5 represent both an error of a factor 2. (The figures may therefore not be immediately intuitive, but we want to depict that these models rather overestimate than underestimate the true bandwidth, i.e., that the relative model error is rather above than below the dashed line.) As can be clearly seen when comparing these graphs, M_3 and M_4 outperform the other two models for connections with a high loss frequency.

Second, Table 8.1 confirms that timeouts are problematic from a modelling perspective. M_2, which does not model timeouts, matches effective throughput worse than M_3 and M_4. M_1 achieves high values for R^2 merely for connections that do not suffer from timeouts. There is a simple way to improve M_1: we can adjust the computation to take the number of timeouts (TO) and the average duration of a timeout (t_{RTO}) into account. More precisely, the average $cwnd$ size is reduced by $\frac{cwnd \cdot TO \cdot (t_{RTO} - rtt)}{t_{RTO}}$. For this variation of M_1 we obtain $R^2 = 0.94$ for all connections. (R^2 then is 0.86 for all lossy connections, 0.77 for lossy connections with timeouts; the other values remain unchanged.) However, even with this adjustment, M_3 and M_4 are clearly more attractive than the other two models.

Third, Table 8.1 also illustrates that we should exercise caution when summarizing the "goodness" of a model by a single number such as R^2. As we learn from the numbers for M_2 for instance, the clusters for "lossy, no timeouts" and "lossy, timeouts" connections each seem to be better described with a linear regression model than the union of the two disjunct sets ("lossy"). In fact, if we cluster and plot the data again according to timeouts, we see effects such as those depicted in Figure 8.1, however, the variation is smaller.

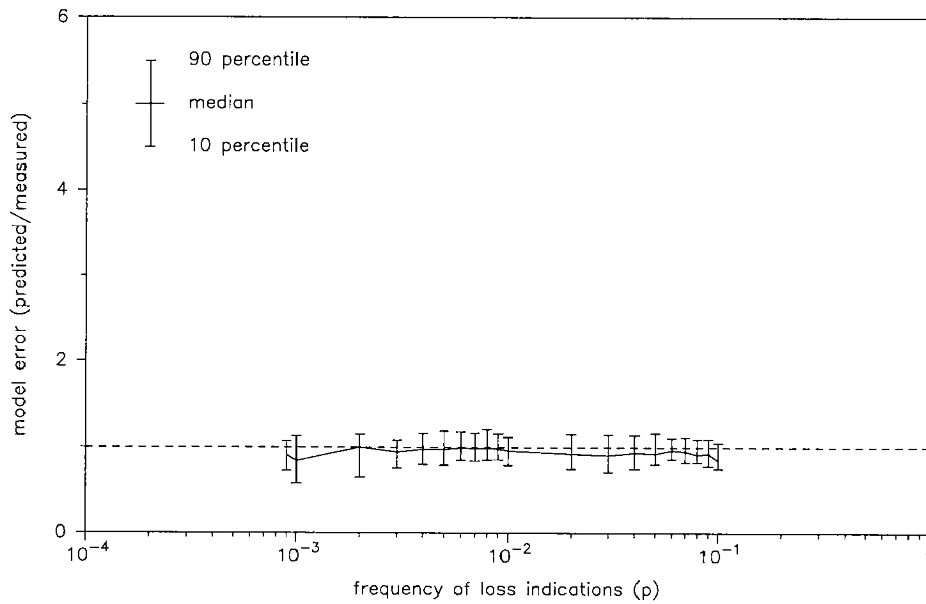


(a) M_1

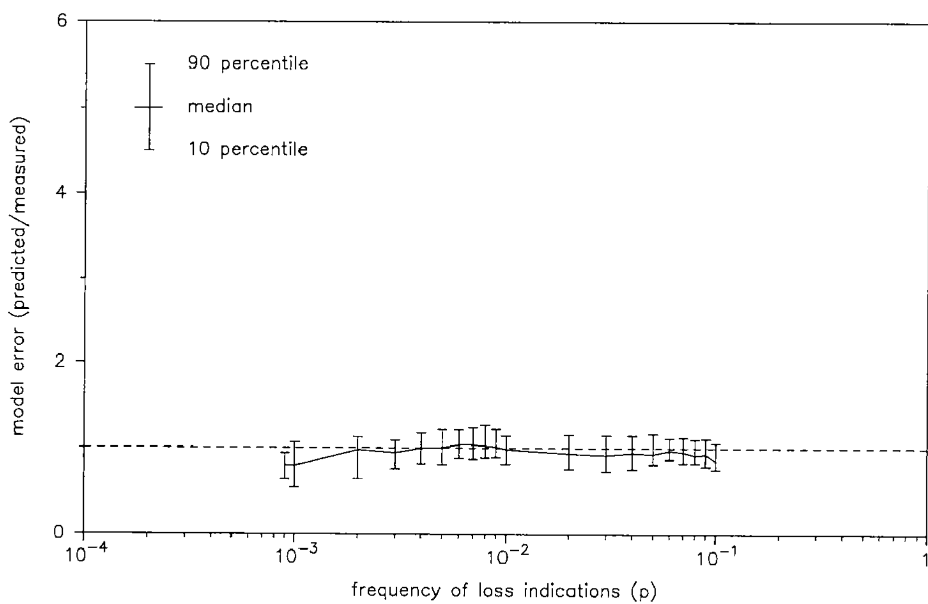


(b) M_2

Figure 8.2: Model error as a function of loss indication frequency (p) for lossy Reno connections.



(c) M_3



(d) M_4

Figure 8.2: Model error as a function of loss indication frequency (p) for lossy Reno connections.

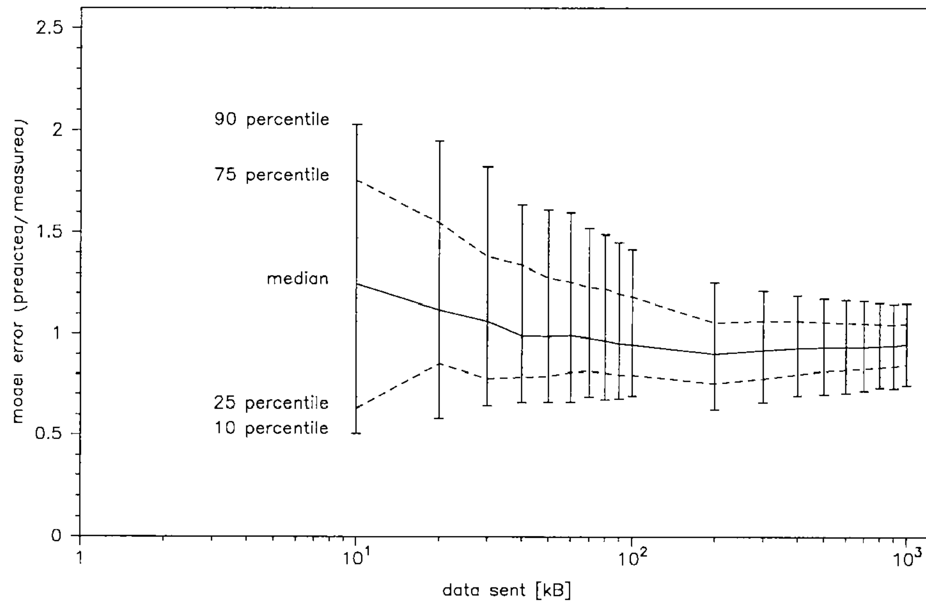
Fourth, we note that according to the R^2 values reported in Table 8.1 M_3 and M_4 seem to model the "lossy, timeouts" cases better than the "lossy, no timeouts" cases. These results are counter-intuitive. Although they are similar for both models, two different lines of arguments are needed to explain the results.

For M_3, we note that loss-afflicted connections that do not experience timeouts generally have smaller loss rates than connections that suffer from timeouts. (We observe a correlation of $+0.93$ between the number of loss indications p and the number of timeouts TO experienced by Reno connections). Furthermore, the higher the loss rate, the shorter the connection stays in the initial slow-start phase, and thus, the more the connection is governed by congestion avoidance. Since M_3 only models congestion avoidance behavior, it is not surprising to see that M_3 better matches the throughput of connections that suffer high loss rates (and timeouts), and are thus more likely to be dominated by congestion avoidance.

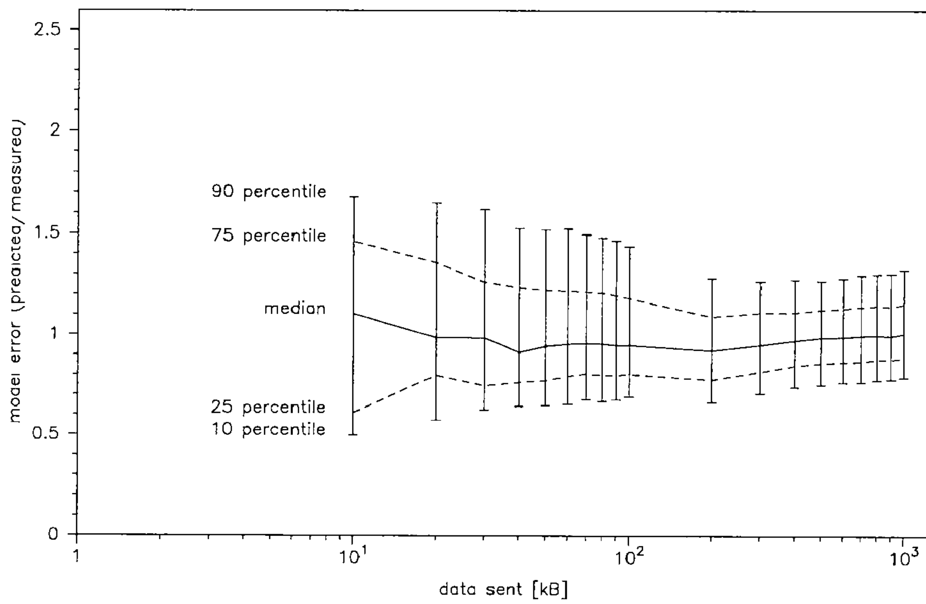
For M_4, different arguments apply. Recall, M_4 is supposed to accurately model the throughput of a TCP connection regardless of how much time the connection spends in slow-start or congestion avoidance phase, respectively. Cardwell et al. [27] note that using a model for steady-state throughput to characterize the time required to transfer the remaining data after slow-start introduces the following error. When the sender detects a loss in the initial slow-start phase, its *cwnd* will often be much larger than the steady-state average *cwnd*. The size of *cwnd* after slow-start can be roughly approximated by $\frac{1}{3p}$ (see [27]). The steady-state value for *cwnd* is $\sqrt{\frac{3}{2bp}}$ (see [108, 136]). Thus, for high loss rates (i.e., for timeout-afflicted connections), the sender exits slow-start at nearly the steady-state *cwnd* value, so the model error should be small. For small loss rates it can take three or more loss indications—corresponding to megabytes of data—to reach steady-state, so M_4 will often underestimate the effective throughput of medium-sized connections, such as those studied in our experiment. (This conclusion is supported by the scatter plot in Figure 8.1 (b), which shows that M_4 often underestimates the bandwidth slightly.)

In summary, we find that M_3 and M_4 perform about the same. For the 1MB transfers we observed, both models have their limitations. Despite these limitations, it is surprising to see how well M_3 and M_4 model the throughput experienced by a TCP connection. For shorter connections, which are more likely to be dominated by the initial slow-start phase, M_4 is likely to win over M_3. However, such connections are less of interest to network-aware applications. For longer connections, the effect of the initial slow-start is negligible, congestion avoidance dominates, and thus both models match the throughput of such connections equally well.

Knowing about modelling errors at the end of a connection is only of limited usefulness to network-aware applications. Such applications need accurate and timely information as early as possible. Figures 8.3 (a) and (b) show how the model errors of M_3 and M_4 decrease as a transfer progresses. For each data point x on the (logarithmically scaled x-axis), we apply M_3 and M_4 to model the throughput experienced by the connection during the first x kB of data sent. The figures report the median, as well as the 10 and 90 percentiles of the relative model errors. By virtue of M_3, only those connections were considered which experienced loss in the first x kB of data sent. Figure 8.3 (a) indicates that M_3 models the throughput fairly accurately even early on in "lossy" connections—which are the most difficult to model—as for 80% of the connections the model differs by less than a factor of 2 (i.e., lies between 0.5 and 2). Figure 8.3 (b) shows how the model error of M_4 for "lossy" connections evolves as a transfer progresses. We find that M_4 is more accurate than M_3 early on in a connection, because



(a) M_3



(b) M_4

Figure 8.3: Model error as function of data transferred (lossy connections).

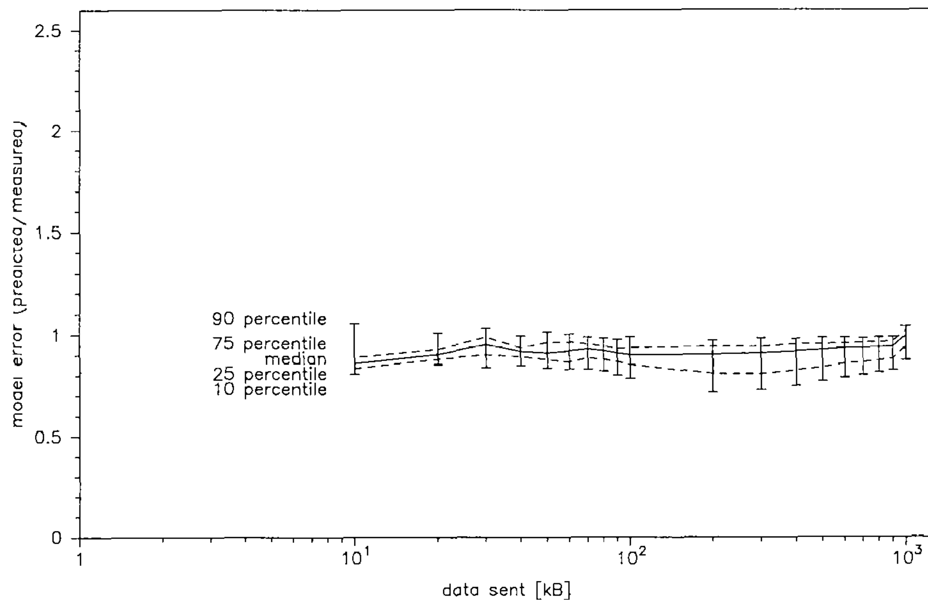


Figure 8.4: M_4 model error as function of data transferred (lossless connections).

M_4 models slow-start behavior. However, the differences are rather modest. Figure 8.4 depicts the model error of M_4 for loss-free connections as they progress. M_4 very accurately models the throughput for such connections, regardless of their progress. This result indicates that M_4 accurately captures TCP behavior even in the early stages of a data transfer.

The results for M_3 and M_4 reported in this section encourage the use of transport-layer information in modelling network performance. These results provide a big incentive to pass the appropriate information to the application (or a framework upon which the application is built) and to widen the operating system's application programming interface minimally to allow an application to take advantage of the fact that such valuable information is readily available in the transport protocol stack. Chapter 9 shows that such an application-protocol cooperation can be implemented with rather small efforts.

8.4 Timeouts

It is widely agreed that retransmission timeouts (RTOs) are problematic for various reasons. Besides causing modelling headaches and therefore unpredictable protocol behavior for network-aware applications, they may waste network resources due to unnecessary retransmissions during slow-start after the timeout. Unfair sharing of the bottleneck and burstiness [125, 97], as well as performance losses experienced by the application [9], are other sources of concern.

Thus, it would be highly desirable if timeouts could be avoided as often as possible. To do so, we must understand the circumstances in which timeouts in TCP Reno happen. Following an earlier classification [97], we distinguish 3 scenarios:

Multiple losses: Multiple packets in a single congestion window are lost, and multiple fast retransmission/fast recovery cycles occur until the flow of duplicate acknowledgments ebbs off, leading to a RTO.

Non-trigger: The sender retransmits a packet without previous attempts because the fast retransmission algorithm has not been triggered.

Lost retransmission: The sender retransmits a packet that has already been sent in a preceding RTO or a fast retransmission.

Over the last few years, numerous TCP enhancements have been proposed that try to avoid timeouts and address the problems mentioned above. Selective acknowledgments (SACK) [107] and FACK (forward acknowledgment) [105, 106] aim at solving the problems caused by multiple packet loss in a single congestion window. The SACK enhancements also allow the repair of lost fast retransmissions [106]. Studies of packet traces [97] have led to the conclusion that multiple packet losses are the source of only a very small fraction of the RTOs encountered by TCP connections and that over 85% of RTOs are due to non-trigger of fast retransmission.

The data presented in the studies cited above were collected in 1995 and 1996. Although our data is not new in the sense that it points out new aspects of TCP behavior, the rapid evolution of the Internet (towards higher bandwidths for many links) introduces enough changes to warrant new experiments. Using a 3-month period of traces, we inspected the retransmission timeouts and their causes for all the Reno connections. We find that the 2486 Reno connections (from this subset) suffered 23279 timeouts, of which 39.5% are due to non-trigger, 40.1% are due to multiple packet loss, and 20.4% are due to lost retransmissions (10.6% lost fast retransmissions, 9.8% lost timeout retransmissions). To get a handle on the benefits of SACKs, we installed receivers generating SACKs for Reno senders (which do not handle SACKs) and then traced the SACKs at the sender. This setup allows us to assess how many of the non-trigger timeouts could have been avoided by exploiting the SACK information (see extended recovery trigger condition [105]). It turns out that in our experiment, 32.9% of the non-trigger timeouts would have been avoidable with SACKs. For the other 67.1% of the non-trigger timeouts, optimizations such as NetReno [97] might have been effective.

To summarize, out of all the Reno timeouts observed, 63.7% might have been avoided with SACK-enhanced protocols[7]. Note that this is an upper bound on the number what *might* be curable with SACKs. In Section 8.5 we report on how effective the SACK enhancements *are* in avoiding retransmission timeouts. In the other 36.3% of the cases SACKs would not have been helpful. For the remaining 26.5% non-trigger timeouts (not avoidable with SACKs), approaches such as NetReno would be applicable[8]. The numbers captured by our experiment largely differ from numbers reported in previous studies: 85% of the timeouts are reported non-trigger situations in [97], and only 4% of the timeouts have been classified as being avoidable by SACKs [9]. These differences can be attributed to the fact that the connections we traced observed bandwidths that are (at least) an order of magnitude higher as those reported in [10] and [141] (which are based on the same data as the studies cited above)[9]. The bandwidth change observed may reflect the general trend towards higher bandwidths but may also be due to the fact that the sites participating in our study are generally well connected to the Internet.

[7] 63.7% = 40.1 (multiple loss) + 10.6 (lost fast retransmissions) + 0.329 * 39.5 (avoidable non-trigger)%.

[8] 26.5% = 0.671 · 39.5 (unavoidable non-trigger)%

[9] Higher bandwidths imply larger congestion windows. Timeouts are less likely to occur if the congestion window is large. The formula $Q(p, b, w)$ introduced by Padhye et al. [136] to compute the probability that a packet loss leads to a timeout exactly reflects this correspondence between timeouts and the congestion window size w (see Section 8.1).

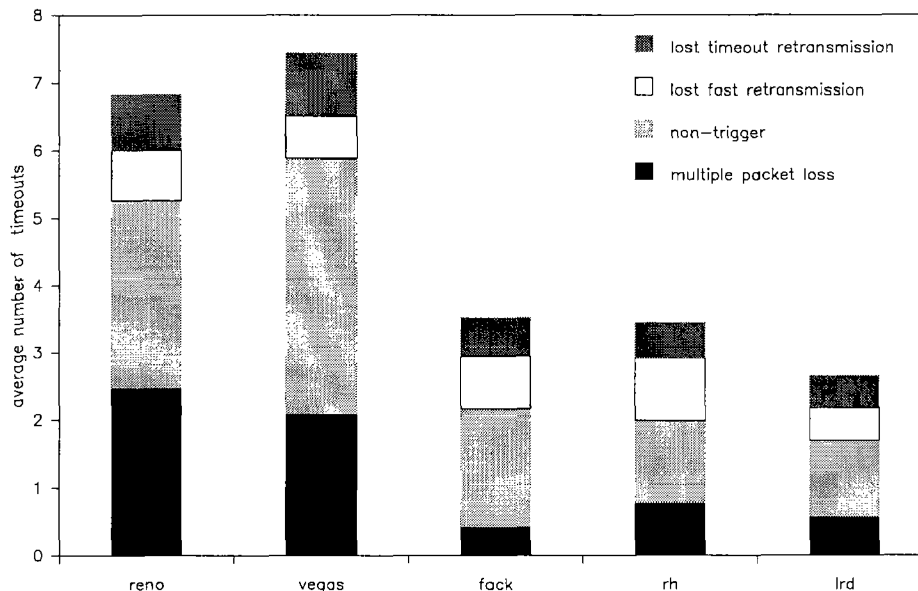


Figure 8.5: Breakdown of timeout causes for each protocol.

8.5 Evaluation of protocols

After assessing the potential benefits of various TCP enhancements with regard to timeout avoidance, we now compare the implementations of these protocol enhancements based on the actual measurements. We first report on the effectiveness of SACK-enhanced protocols in avoiding timeouts and then resume our analysis of throughput models to see the effect of using different protocol implementations.

8.5.1 Effectiveness in avoiding timeouts

Figure 8.5 shows how the different protocols are effected by timeouts. The heights of the bars depict the mean number of timeouts per connection, where the average is based on all connections using a given protocol. Additionally, the figure shows a detailed breakdown of the causes for these timeouts (according to the classification given in Section 8.4). While a detailed comparison of the different protocols is beyond the scope of this dissertation (see [19] for details), we note that the SACK-enhanced protocols are significantly more effective in avoiding timeouts than Reno. On the other hand, Vegas underperforms Reno as far as timeouts are concerned.

To simplify the presentation, and because the three SACK-enhanced protocols are similarly effective in avoiding timeouts, we use FACK as a placeholder for other SACK-enhanced protocols. For similar reasons we do not further discuss Vegas. Thus, we focus on the comparison of Reno and FACK connections. Overall, we find that FACK reduces the number of timeouts experienced considerably. 67.9% of the Reno connections suffer at least one timeout. The mean number of timeouts for Reno connections is 6.8. Only 34.1% of the FACK connections are timeout-afflicted. The mean number of timeouts for FACK connections is 3.5, which is a 48.3% improvement over Reno. Corresponding to the three classes of timeouts, there are three mechanisms present in SACK-enhanced protocols to avoid timeouts. We look at their effectiveness in turn (see [19] for details):

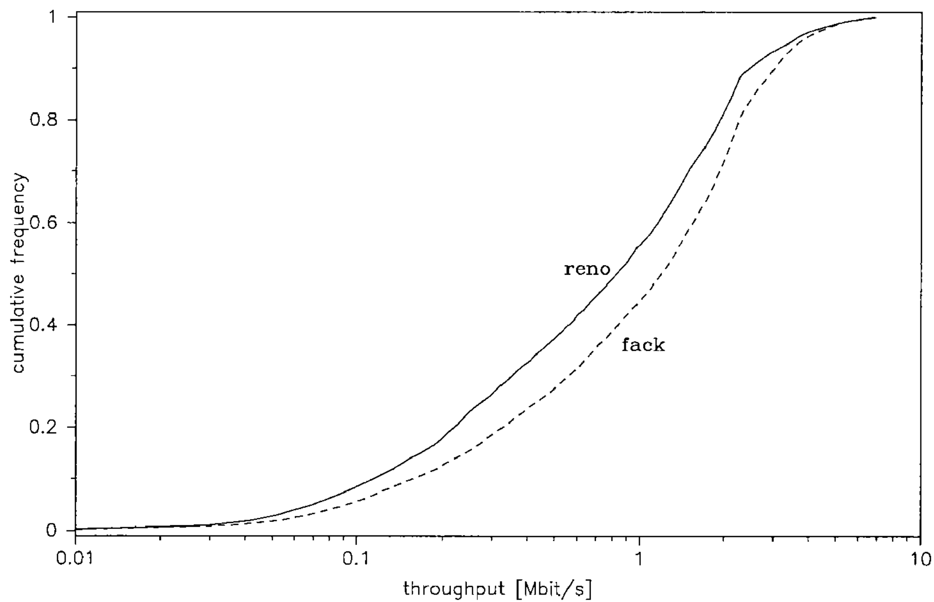


Figure 8.6: Comparison of throughputs for Reno and FACK connections.

Multiple loss: When looking at timeouts due to multiple packet loss, we find that compared to 65.4% Reno connections only 18.3% FACK connections suffer from such timeouts, and the mean number of such timeouts per connection drops from 2.5 (Reno) to 0.4 (FACK). We conclude that FACK can drastically cut down the number of timeouts due to multiple loss. By and large, this reduction is achieved by the use of SACKs; the contribution of the other optimizations in the FACK algorithm is only minor in this respect.

Non-trigger: The FACK algorithm extends Reno's recovery trigger condition beyond the duplicate acknowledgment threshold logic to better cope with multiple packet loss and loss of duplicate acknowledgments, which are particularly problematic for connections with small *cwnd* [105]. We find that 38.3% of the Reno connections suffer from at least one non-trigger timeout. The mean number of non-trigger timeouts for Reno connections is 2.8. Only 19.2% of the FACK connections suffer from at least one non-trigger timeout. The mean number of non-trigger timeouts for FACK connections is 1.8. The difference in means closely matches the reported fraction (approximately one third) of non-trigger timeouts that were termed "avoidable by SACKs" in Section 8.4 (the difference is due to the fact that only a subset of the traces is used to assess the potential benefits of SACKs).

Lost retransmissions: Both Reno and FACK can recover from a lost retransmission only by a timeout. Lost retransmission detection [106] extends the FACK algorithm to avoid such timeouts; it achieves a 40% improvement over FACK.

8.5.2 Effect on throughput

Figure 8.6 provides insight into how FACK's ability to avoid timeouts relates to performance seen at the application level. FACK performs consistently better than Reno as can be seen from the clear "right shift" of the bandwidth distribution curve. (See Figure 2 in [19] for a comparison that includes the other TCP variants evaluated.)

R^2		Reno				FACK			
		M_1	M_2	M_3	M_4	M_1	M_2	M_3	M_4
all connections		0.74	–	–	0.96	0.98	–	–	0.95
no loss	$(p = 0)$	0.99	–	–	0.97	0.99	–	–	0.96
lossy	$(p > 0)$	0.62	0.83	0.92	0.91	0.97	0.91	0.93	0.93
lossy, no timeouts	$(p > 0 \wedge TO = 0)$	0.99	0.85	0.85	0.86	0.99	0.91	0.91	0.90
lossy, timeouts	$(TO > 0)$	0.63	0.86	0.93	0.93	0.97	0.93	0.93	0.94

Table 8.2: Summary of models (Reno and FACK connections).

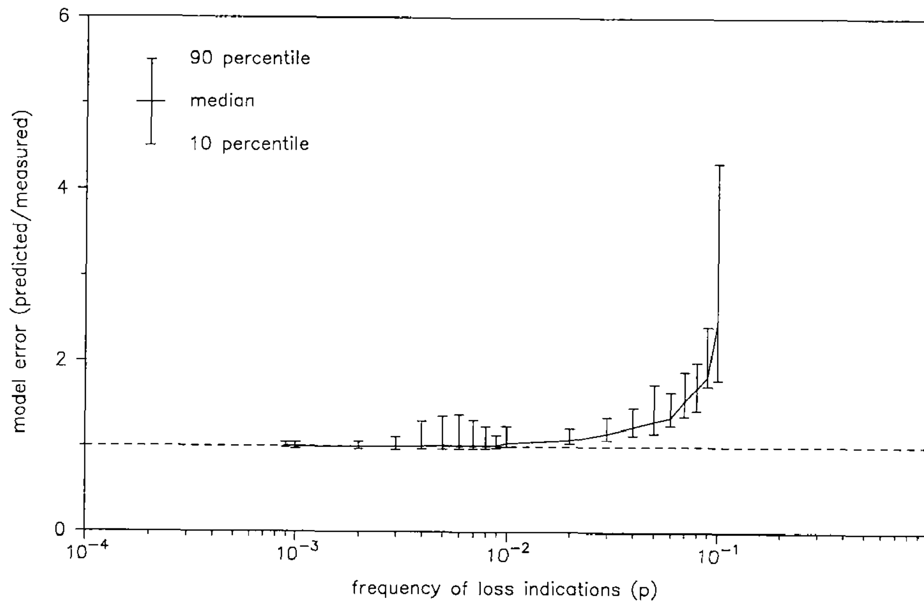
R^2	Reno	Vegas	FACK	RH	LRD
M_1, all connections	0.74	0.67	0.98	0.99	1.00
M_3, lossy connections	0.92	0.70	0.93	0.92	0.92
M_4, all connections	0.96	0.84	0.95	0.93	0.93

Table 8.3: Comparison of models and protocols.

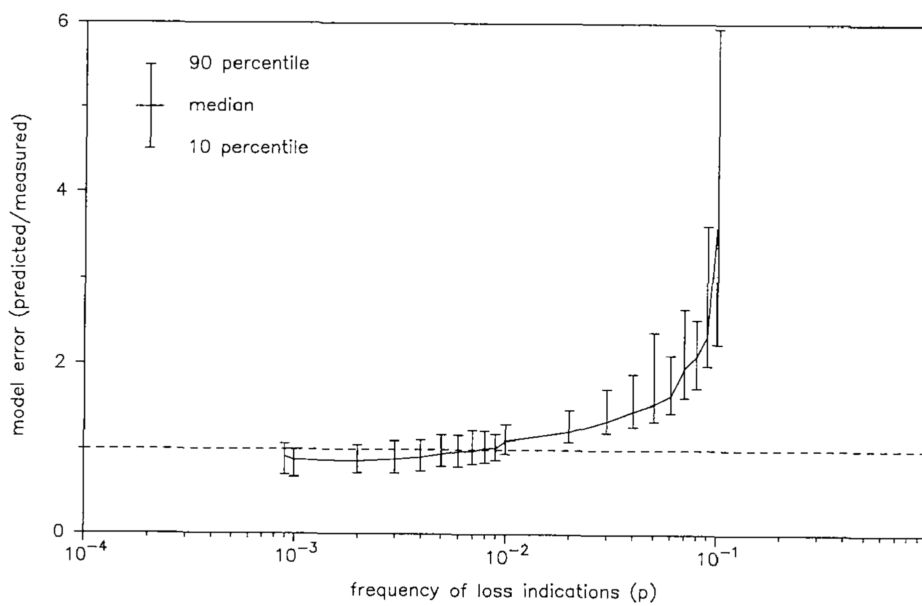
8.5.3 Effect on throughput models

Following the methodology used in Section 8.3, we try to assess the quality of the four models M_1, M_2, M_3 and M_4 for FACK connections by fitting the data with a least-squares fit and by computing the coefficient of determination R^2. Table 8.2 repeats the results for Reno from Table 8.1 and contrasts them with the results obtained for FACK connections. Again, p denotes frequency of loss indications, and *TO* refers to the number of timeouts per connection. There are two noteworthy aspects: First, the simplistic model M_1 "performs" much better for FACK than for Reno connections. Second, the differences in the "quality" of the models are much less pronounced for FACK than they are for Reno. This improvement can mainly be attributed to FACK's success in avoiding retransmission timeouts. The differences in terms of timeouts noted between the protocols (see Figure 8.5) are closely reflected by the accuracy of the TCP models for these protocols (see Table 8.3). The models perform worse for Vegas than for Reno. One of the reasons is that Vegas connections generally suffer more timeouts than Reno connections (see Figure 8.5). Furthermore, the models perform about the same for all SACK-enhanced protocols. These protocols experience about the same number of timeouts.

This reasoning is further supported by the model error plots for lossy FACK connections shown in Figures 8.7 (a)–(d). Comparing these to the Figures 8.2 (a)–(d) allows us to see the effect of using SACK-enhanced protocols. Again M_3 and M_4 perform best over the entire range of loss rates witnessed. In general, the clusters of the model errors appear to be tighter with fewer outliers (particularly for M_1). This improvement is reflected in the increase in R^2 as reported in Table 8.2. However, the models also exhibit basically the same behavior for high loss probabilities (which occur mainly in connections suffering from timeouts). The reasons why (i) R^2 for FACK connections is higher than for Reno connections, and why (ii) R^2 differs only slightly for M_1 and M_3/M_4 in the case of FACK (as opposed to Reno) can be found in the distributions of the loss indication probability for the two protocols, plotted in Figure 8.8. The log-scaled x-axis represents the range of loss indication rates encountered (to be consistent with the data in Figures 8.7 (a)–(d), we only consider lossy connections). Reno connections have a consistently higher loss indication probability than FACK connections. This property explains

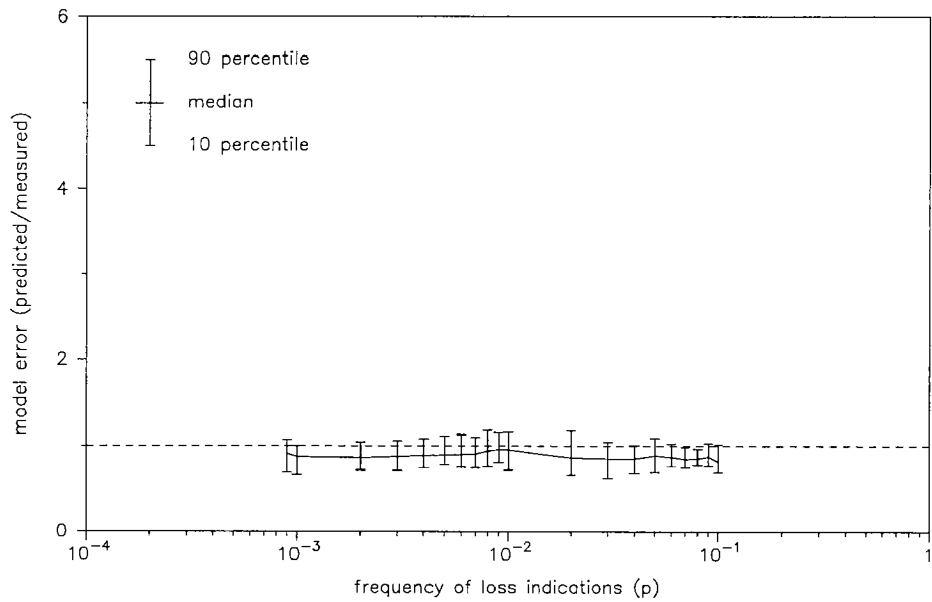


(a) M_1

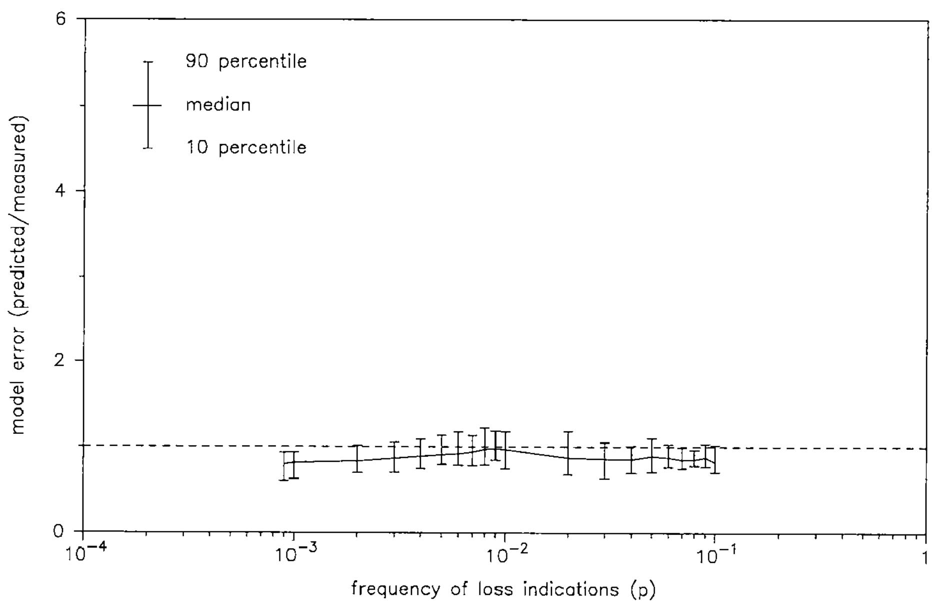


(b) M_2

Figure 8.7: Model error as a function of loss indication frequency (p) for lossy FACK connections.



(c) M_3



(d) M_4

Figure 8.7: Model error as a function of loss indication frequency (p) for lossy FACK connections.

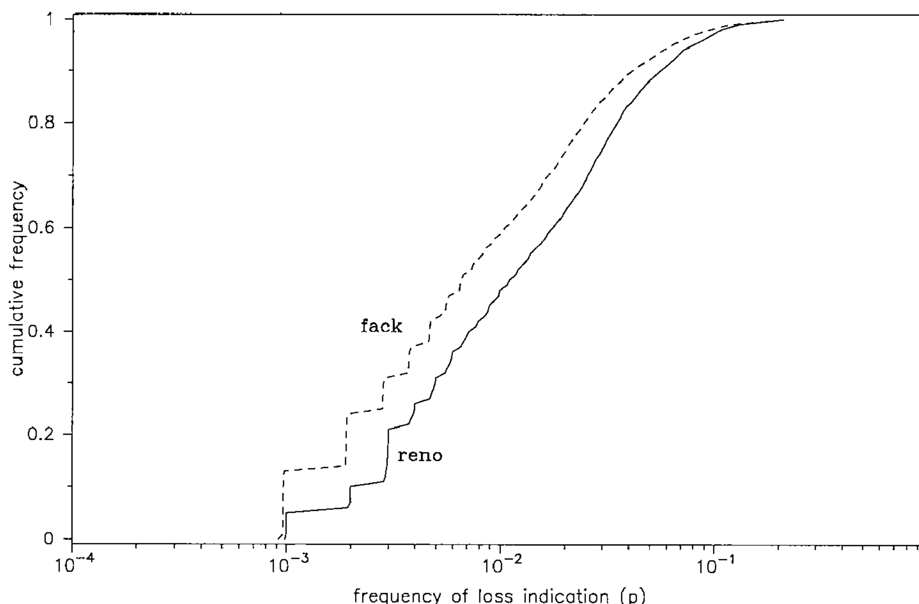


Figure 8.8: Distribution of loss indication probabilities for lossy Reno and FACK connections.

(i). The increase in modelling error for M_1 around $p \approx 0.1$ is less pronounced in the case of FACK connections (and therefore the difference between M_1 and M_3/M_4 is smaller), because fewer FACK connections experience such loss rates than Reno connections.

The loss indication probability for FACK connections is lower than the one for Reno for two reasons: (i) the marked decrease in the number of timeouts and (ii) the fact that FACK (like other SACK-enhanced protocols) treats the loss of multiple packets within one congestion window as one event (compared to standard Reno, where multiple-packet congestion signals lead to multiple recoveries and hence multiple reductions of the congestion window).

In summary, the results presented in this section add another dimension to the conclusions of earlier studies [108, 136], which show how accurately the different throughput models match the bandwidth experienced by a connection. We find that the TCP variant used by a connection has a non-negligible impact on the accuracy of the models used for bandwidth estimation.

8.6 Stability

Since M_3 and M_4 yield reasonable results, it is interesting to turn to an investigation of the temporal stability of available bandwidth. Furthermore, it is interesting to note whether and how our empirical data compares to those presented in earlier studies [141, 10] (see Section 7.2.2).

A preliminary analysis of our experimental data seems to support the findings of these previous studies (however, for smaller time-scales: $10^0 - 10^2$ seconds): We first look at how well the model for the first 50% of the data transferred predicts the performance observed for the second half of the connection. Figure 8.9 shows the cumulative distribution function of the factor of bandwidth change observed (which in our simple prediction scenario is equivalent to the relative error of the prediction). We find that 84% of the Reno connections experienced a change of less than a factor of 2 (to the better or the worse). Note that the figure does not show the complete distribution, as approximately 1% of the connections experienced a throughput

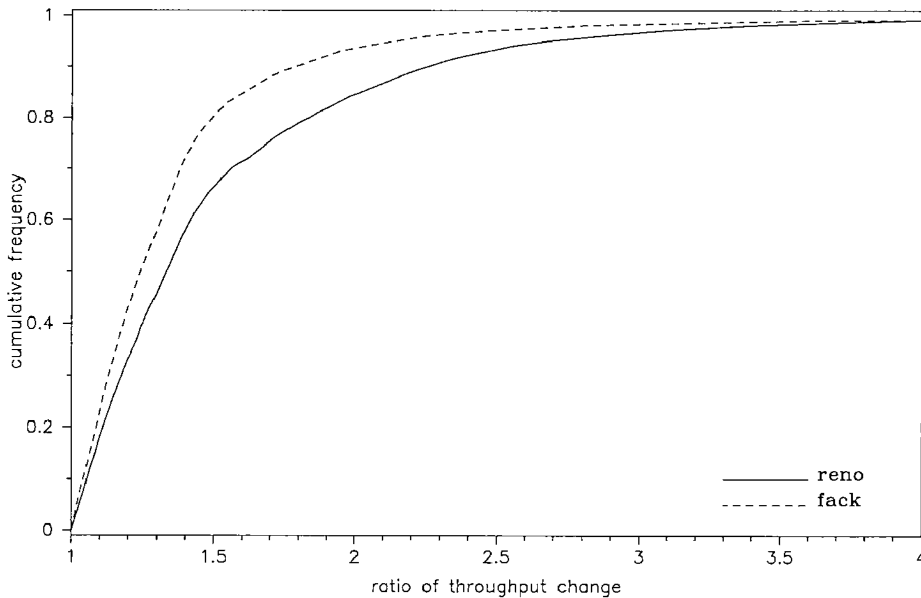


Figure 8.9: Throughput change between first half to second half of connection.

change of more than a factor of 4 (up to a factor of 100!).

Clearly, this simplistic view on bandwidth stability is limited and does not parallel the more rigorous and well-founded analyses of other researchers [141, 10, 186]. Nevertheless, it allows us to confirm that a significant fraction of the connections (in our experiment) experience fairly stable bandwidths. However, the heavy-tailed distributions of bandwidth change shown in Figure 8.9 also indicate that there is a non-negligible fraction of connections which experience widely fluctuating bandwidths. Thus, if we keep in mind that our experiment included only well-connected hosts, we can conclude that bandwidth monitoring is (and will continue to be) important for network-aware applications that want to achieve predictable application behavior.

Figure 8.9 allows to draw a second conclusion: FACK connections experience smaller changes in bandwidth between the first and the second half of a connection than Reno connections as can be seen from the "left shift" in the cumulative distribution for the ratio of throughput change. This finding implies that not only the ability to model available bandwidth may be positively influenced by the transport protocol used, but that the TCP variant also can have a notable effect on the ability to predict bandwidth availability in the future.

Network-aware applications need predictions about the bandwidth available in the future. Such predictions must be based on a varying amount of past observations, and may well be required to predict the performance for varying time-scales into the future. Thus we are interested in assessing how the accuracy of a prediction depends on the amount of "history" data observed and the "future" to be predicted. Figure 8.10 shows (on the y-axis) the ratio of bandwidth change (or relative prediction error) as a function of the "history" observed and (on the x-axis) the "future" extrapolated. We show the median, 80 and 90 percentiles of the prediction error for each combination of the "history" and "future" values chosen[10]. The figure reflects two obvious

[10]For each point in the trace and each combination of the parameters "history" and "future" applicable, we determine the relative prediction error and aggregate all the values for a "history"/"future" pair. Note that we exclude the slow-start from the "history" data. Slow-start's transient behavior dilutes findings about TCP steady-state behavior significantly and requires separate treatment [27].

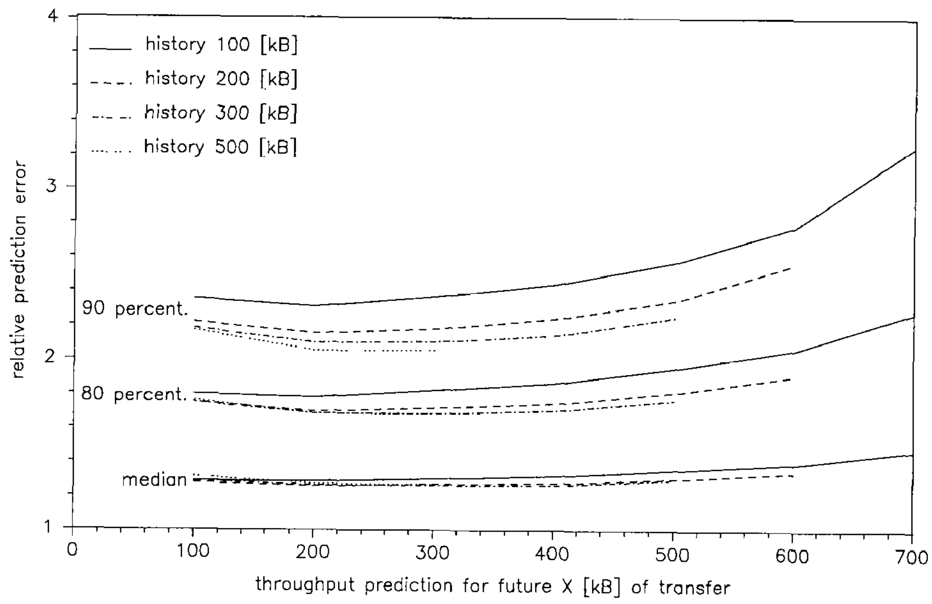


Figure 8.10: Throughput change as function of history and future.

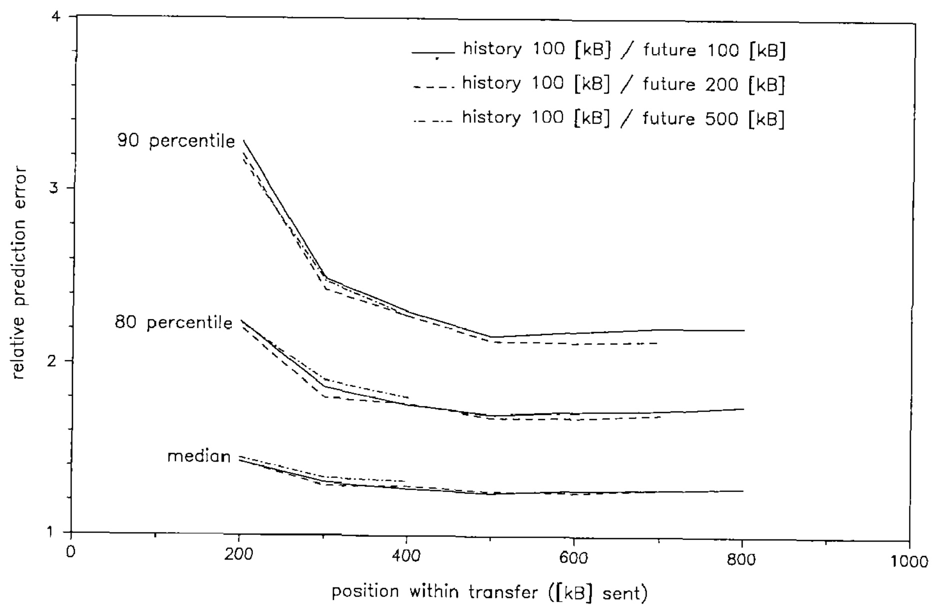


Figure 8.11: Throughput change as function of history, future, and position within connection.

facts: first, the more history data is used for the prediction the smaller the prediction error, and second, the further into the future we want to predict the less accurate the prediction is. What is more surprising to see, however, is that for most connections (i.e., 80%) the dependence on the "history" and "future" parameters seems to be fairly small. This result indicates that even with medium level of past information, reasonable predictions about future performance seem to be possible (at least for the connections we observed).

Furthermore, it is interesting to note that the prediction errors (stability of the throughput) observed seem to depend on the position within a connection. Figure 8.11 plots the same data for the prediction error for a fixed parameter "history" (100kB) and different levels of "future" as a function of the position within a connection. Again, we excluded slow-start from the "history" data. We find that the prediction error decreases as the connection progresses (towards steady-state). In principle, this result indicates that more work is required to understand the dynamics and performance implications of slow-start with respect to network-aware applications. However, as shown by Cardwell et al. [27], the slow-start performance depends mainly on the probability of packet loss. Hence, as far as bandwidth monitoring is concerned, we are caught in a catch-22 situation, since the ability to predict throughput (at the beginning of a connection) depends on the ability to predict packet loss rates, and the occurrence of packet loss depends on the application demands (in terms of bandwidth) placed on the network.

8.7 Summary

Adaptive applications need accurate and timely information about the currently available bandwidth. Our evaluation of four simple models to compute the bandwidth, based on a collection of detailed traces collected in the course of a 6-month Internet experiment, allows us to comment on three aspects. First, these simple throughput models are able to characterize the bandwidth available to a single TCP connection fairly accurately. Although the original design goal was to obtain the asymptotic bandwidth, the estimators M_3 and M_4 perform well enough to be considered for network-aware applications. The effectiveness of M_3 and M_4, which are based on transport-level parameters, provides an incentive to allow an adaptive application (or the framework that is extended by the application) access to these values in lower protocol layers. Such access must be without overhead so that M_3 and M_4 remain cheap to compute and so that the information is available in a timely fashion. Second, our data also add another dimension to the results of earlier studies that reported the benefits obtained from various enhancements to TCP. We find that the transport protocol (i.e., the TCP variant) used has a noticeable impact on the accuracy of models used for bandwidth estimation. These TCP enhancements reduce the number of timeouts and thereby improve, as a side effect, the accuracy of the simple models that we investigated. Third, our limited analysis of issues related to bandwidth stability showed that—even though bandwidth seems to be fairly stable for a large fraction of the connections observed—significant bandwidth fluctuations can be observed. From these observations we can draw two conclusions. First bandwidth monitoring is (still) important for network-aware applications. Second, a significant fraction of the Internet traffic is sufficiently stable so that bandwidth prediction seems possible. Moreover, we note a positive effect of SACK-enhanced protocols on the predictability of available bandwidth.

Chapter 9

Comparison of monitoring approaches

Chapter 7 describes different approaches to bandwidth monitoring and classified them according to their method of information collection. The qualitative comparison could not conclusively answer which of the two approaches, application-level or transport-level monitoring, is better suited to the needs of network-aware applications. The evaluation of different TCP models in Chapter 8 shows that transport-level monitoring is a promising approach (because the models are fairly accurate) and indicates that a simple widening of the transport protocol API may suffice to provide a network-aware application with the required information. However, some questions are still unanswered. How do the two approaches of application-level and transport-level monitoring compare quantitatively in terms of the metrics identified in Chapter 7 (efficiency and quality of the bandwidth information)? Moreover, it is unclear how difficult it is to actually implement a transport-level monitor. How many changes to the protocol stack are necessary?

To answer these questions, the different approaches must be implemented. This chapter presents the architecture of our bandwidth monitoring prototype (Section 9.1), describes the implementation of the two approaches to information collection (Section 9.2), and finally gives a quantitative evaluation and comparison of the two approaches (Section 9.3).

9.1 Monitor design

This section discusses the design of our prototype monitoring toolkit. Section 9.1.1 briefly lists the requirements for the software architecture. Section 9.1.2 gives a high-level overview of the system. Section 9.1.3 sketches the interface to the monitoring system, and Section 9.1.4 presents the processing steps involved in producing bandwidth estimates.

9.1.1 Requirements

In addition to the requirements stated in Chapter 7, the software architecture of our monitoring toolkit should ideally have the following properties:

Integrated. There are three aspects to note. First, to allow for an unbiased comparison of the overheads incurred by the different approaches to information collection, these approaches should be *integrated* in a single monitoring architecture. By integrating and

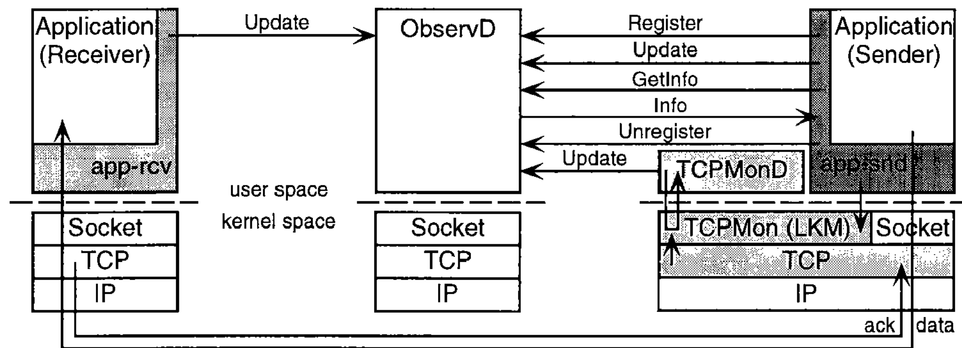


Figure 9.1: Simple monitor architecture that integrates different approaches to information collection.

unifying the process of information collection and bandwidth modelling in a single architecture, we ensure that our conclusions about the efficiency of the different monitoring techniques are not disturbed by differences in the implementation. Second, to be able to compare the different approaches with respect to the quality of the bandwidth information provided to the application, the system must allow a connection to be monitored with multiple approaches *simultaneously*. And third, independent of the approach used to collect bandwidth information, the application should be presented with a *unified* programming interface (so that only minimal changes are necessary to the application to experiment with different approaches).

Extensible. It should be easy to add new techniques of information collection to the monitoring system. In addition, the system should be extensible so that new bandwidth models (e.g., new TCP throughput models) for existing information collection techniques can be added. Furthermore, it should be easy to augment the monitoring toolkit with prediction services, such as those provided by the RPS toolkit developed by Dinda et al. [43].

Relocatable. It should be possible to place the (resource-intensive parts of the) monitoring system on other machines on the network. Bandwidth modelling and on-line prediction [43] can incur significant computational load (in particular, if numerous connections must be monitored simultaneously). Therefore, it should be possible to relieve, e.g., a busy network-aware server, from this load.

9.1.2 Architecture

Inspired by the distinction between collectors and modellers made by the designers of the Remos system [99, 121], we pursue a two-tier approach in our monitor architecture. We refer to components that are responsible for network-oriented functionality, that is, for the collection of raw network status and performance samples as "sensors" (collectors in Remos). The "observer" component (modeller in Remos) is responsible for application-oriented functionality, e.g., aggregating performance information of connections sharing a network path, or forming bandwidth estimates from the raw performance samples. Figure 9.1 depicts an overview of the components involved in our monitor architecture.

Observer. In our prototype, an observer component is realized as a daemon (*observd*) that can be run on any host in the network. E.g., an observer can (but need not) be co-located with the (sending) network-aware application that requests bandwidth information. Co-locating the observer with the application has the advantage of small communication overheads. The observer daemon manages and caches the state and performance information of multiple active connections simultaneously. The daemon accepts and processes various types of messages, e.g., messages that indicate the "arrival" or the "departure" of a connection (register and unregister), deliver performance updates (from the sensors), etc. These messages can originate from different sources (applications and sensors). The observer can be queried both for bandwidth estimates for specific connections that are currently active, or for estimates on the aggregate bandwidth of multiple connections between two hosts. The daemon is started by inetd (Internet super-server) when any message arrives, and it terminates after having been inactive for some amount of time. Multiple applications can (but need not) share an observer component. The interaction with and the operation of observer components is described in more detail in Sections 9.1.3 and 9.1.4.

Sensors. Sensors are conceptually simple components that must collect raw performance data, encapsulate the data in update messages, and send the messages to the observer that processes the performance updates. There are different types of sensors that collect raw, network-oriented performance data. There is one type of sensor for each alternative for information collection. In Figure 9.1, the different types of sensors are highlighted with different degrees of shading. Application-level, sender-based information collection is performed by the component denoted as *app-snd* (dark grey). Application-level, receiver-based information collection is done by the *app-rcv* component (medium grey). Transport-level information collection is accomplished by the *tcpmon* component(s) (light grey). Transport-level monitoring must be co-located with the sending application. Section 9.2 describes the design of these sensors in more detail.

This separation of observer and sensor components achieves the desired integration of and extensibility with respect to different information collection mechanisms. A sensor must merely implement the protocol for performance updates. Moreover, relocating the functionality of bandwidth modelling and prediction (implemented by the observer components) is straightforward.

Soft state-based communication

Not being critical for the correct operation of network-aware applications, control information (performance reports, status messages, or connection advertisements) can be transmitted between sensors, applications, and observer(s) using an unreliable datagram transport service such as provided by UDP [176]. With such a transport service, a message (datagram) may either be delivered correctly to the recipient (i.e., complete and ungarbled), or it may not arrive at all.

Unreliable delivery is the appropriate mode of transport for performance reports (update messages). It makes little sense to retransmit performance reports that are dropped by the network because they may have been superseded by new reports in the meantime. In the interest of accuracy, the performance reports should nonetheless be communicated in a loss-resilient fashion, such that each update message is self-contained, that is, allows the observer to model

the available bandwidth accurately. The loss of messages that advertise "arrivals" and "departures" of connections to be monitored is slightly more problematic, because these messages are responsible for the initialization and deletion of connection state in the observer, respectively. Loss of these messages can compromise consistency.

We rely on the concept of *soft state*-based communication that has been successfully applied in other protocols (see [36, 150] for an overview). State is called "soft" if it is maintained by some entity in the network that expires the state after a certain time interval unless it is refreshed by some update message received across the network from another entity in the system. That is, a source of *soft state* (e.g., a sensor in our architecture) transmits periodic "refresh messages" over a (lossy) communication channel to one or more receivers that maintain a copy of that state (observer). Associated with this state is a pending timer, which is reset upon receipt of each refresh message. If the timer expires (because the refresh messages cease), the state is deleted. In our system state is associated with each monitored connection. The state captures whether the connection is active as well as the connection's history of performance/status reports.

The main advantages of soft state-based communication (in comparison to a hard state-based approach) are its simplicity and robustness. Failure treatment and other exception conditions (that often lead to complex, error-prone interactions among many different distributed components in a hard state environment) need not be explicitly engineered, but are implicitly defined by the design of the protocols. In a soft state framework, the designer is forced to presume inconsistency from the start. The following section details the application protocol and describes how the soft state-based communication is realized.

9.1.3 Protocol

The observer protocol specifies how the other components (sensors and applications) communicate with the observer (and thus unifies the different approaches to information collection). The observer daemon accepts and acts upon four types of messages (see Figure 9.1):

Register. A register message specifies a new connection to be monitored, as well as the methods of information collection, bandwidth modelling, and prediction to be used to form bandwidth estimates. A connection is identified by a 4-tuple containing IP addresses and port numbers of both source and destination. The IP addresses describe a host pair, and the two port numbers characterize an individual connection between a pair of hosts. As indicated in Figure 9.1, three alternatives of information collection can be currently used (*app-snd*, *app-rcv*, or *tcpmon*). Any subset of these techniques can be employed simultaneously. For each of these alternatives, the preferred technique for bandwidth modelling and bandwidth prediction can be specified. Several bandwidth models (see Chapter 8) and predictors (see Dinda et al. [43]) may be applicable. It is important to note that information collection and bandwidth modelling are coupled, however, bandwidth prediction is orthogonal to collection and modelling.

Update. Sensors periodically transmit new performance reports for a monitored connection by means of update messages. An update message identifies the connection monitored (by the 4-tuple) as well as the method of information collection employed by the sensor (*app-rcv*, *app-snd*, or *tcpmon*). Furthermore, it carries the raw, uninterpreted performance data produced by

the sensor. The bandwidth model associated with the particular information collection method is responsible to interpret the raw data (see Section 9.1.4). As stated above, this performance data must be provided in a loss-resilient fashion, so that the loss of performance reports does not distort the conclusions about the bandwidth available to the connection. For this purpose, we use cumulative performance metrics that capture what happened since the start of the connection instead of metrics that only report what happened since the last performance report was sent. Examples of such cumulative performance metrics include the total number of bytes sent/received, the total number of timeouts, etc. Cumulative metrics are associated with a timestamp (to allow metrics such as bandwidth or loss rates to be inferred).

GetInfo. An application that requires estimates for the bandwidth available (or for other metrics) can query the observer by means of a getInfo message. The message contains the connection or the host pair as well as the performance metric the requester is interested in. If port numbers are not specified the aggregate performance (bandwidth) for all monitored connections for this host pair are reported. Optionally, the requester can specify which modelling and prediction techniques should be used to produce the estimates. The observer replies with an info message containing the information asked for.

Unregister. By means of an unregister message, an application can inform the observer that a connection has terminated or that it is no longer interested in performance data for the connection.

What happens if any of these messages is lost? If a *register* message is lost, the initialization of the corresponding data structures at the observer cannot take place. This is not problematic, because as soon as update messages for this connection arrive, the lookup of the connection fails and the observer can invoke a (late) initialization of the data structures. (Recall, each update message conveys information about the connection to monitor and about the method used for information collection). Once update messages from each of the sensors employed for a particular connection have been received, the soft state carried by the lost register message is restored. The only information lost with a register message are the application's preferences about the modelling and prediction techniques to use for a particular method of information collection. (This situation may not be dramatic, if meaningful default behavior has been specified.) Each *update* message "refreshes" the state of the corresponding connection. Thus, the loss of update messages is not problematic as it only results in connection states that reflect untimely (but potentially still accurate) performance information. The loss of *unregister* messages is compensated by a timer mechanism that detects whether a connection has been inactive for some time. A connection is inactive if no update messages have been received for the particular time frame. Thus, a lost unregister messages only implies that the memory consumed by the connection state cannot be freed (and written to disk) as early as otherwise possible. If *getInfo* or *info* messages are lost, the application can either resubmit the request or use old information.

In accordance with the messages exchanged between applications and the observer, the monitor's programming interface exposed to applications that want to make use of the bandwidth monitor provides methods to register and unregister a connection to be monitored, specify the mechanisms for information collection, bandwidth modelling, etc., and to query the observer

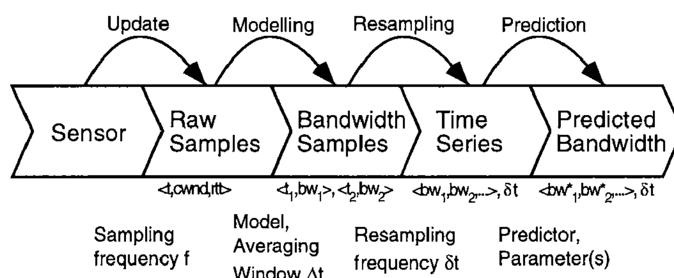


Figure 9.2: Data flow through the processing pipeline.

for bandwidth estimates. This functionality can easily be encapsulated in wrapper functions for the standard socket API calls.

9.1.4 Processing steps

Figure 9.2 depicts the steps involved in producing estimates for the bandwidth available to a particular application. As explained above, sensors collect performance data that is reported to the observer by means of update messages. The sensors typically (to be detailed in Section 9.2) produce measurements periodically with some sampling frequency f. The raw measurements (raw samples) are stored at the observer and serve as the input for the process termed bandwidth modelling (see Chapters 7 and 8). Bandwidth modelling transforms a stream of raw measurement samples into a stream of bandwidth samples. The bandwidth samples depend on the model used and the averaging window Δt used to compute the bandwidth. Because the bandwidth samples may not arrive at the observer with a constant rate, the stream of bandwidth samples may have to be resampled to form a time series (with a fixed interval δt; typically $\delta t \approx 1/f$). Standard methodology for statistical prediction [21, 43] often requires a time series (of bandwidth samples) as input and produce a series of bandwidth predictions as output that can then be reported to an application that queries the observer.

It is important to note that the processing pipeline presented in Figure 9.2 provides a "framework" that integrates and unifies different approaches to each of the individual steps. The framework is extensible in three dimensions. First, new sensors can be added; they must merely conform to the update protocol specified in the previous section. Second, even though a bandwidth model is coupled to a particular method of information collection (it must be able to interpret the raw measurement data), there may be different approaches to interpret the measurement data (cf. TCP throughput models). Third, there exist many different techniques for prediction. E.g., the RPS toolkit implemented by Dinda et al. [43] provides three simple predictors (mean, last, and windowed mean), four predictors that follow the approach proposed by Box-Jenkins [21] (autoregressive (AR), moving average (MA), ARMA, and AR integrated MA (ARIMA)), and one predictor for time series that exhibit long-range dependence (AR fractionally integrated MA (ARFIMA)). Which of these (and possibly other) predictors is best suited for the task of bandwidth prediction for network-aware applications depends on the predictability of network traffic (see Section 7.2.2) and remains to be investigated in future work.

As a consequence, our current implementation does not include this last step of bandwidth prediction, and we leave it to future work to incorporate a prediction system such as Dinda's RPS [43] into the observer component.

9.2 Information collection

This section describes the implementation of the different methods for information collection.

9.2.1 Application-level monitoring

Application stubs

Information collection for application-level monitoring is achieved by wrapping the functions of the socket API. So-called application stubs provide the appropriate wrapper functionality. Sender-based monitoring relies on a sender stub that wraps the send socket calls (*app-snd* in Figure 9.1). Receiver-based monitoring uses a stub that wraps the receive socket calls (*app-rcv*). These wrapper functions update the total count of bytes sent or received and record a timestamp. If the time passed since the last performance report (update message) has been sent to the observer exceeds $1/f$, where f is the sampling frequency, then a new performance report is generated containing the timestamp, the cumulative number of bytes sent or received, and the sequence number of the performance report.

Bandwidth modelling

The update messages sent by application-level sensors basically provide the observer with a sequence number plot of the monitored connection (see Figure 6.5 for an example of such a plot). Bandwidth samples are obtained from these sequence number plots by moving an averaging window of size Δt over the raw samples and by assessing the number of bytes transferred during this interval. The number of bytes transferred Δd is given as the difference of the highest sequence number reported at the end of the window and the lowest sequence number reported at the beginning of the time frame. The bandwidth for the interval is then $\Delta d / \Delta t$. The stream of bandwidth samples is resampled with frequency $1/\delta t$. The resampling is achieved by advancing the averaging window Δt stepwise by δt.

9.2.2 Transport-level monitoring

Information collection for transport-level monitoring is implemented by a layered architecture. The three layers of the approach (see Figure 9.1) include a few hooks in the TCP stack, a loadable kernel module (LKM) termed *tcpmon*, and a user-level daemon process (*tcpmond*). The rationale behind this layered construction is that we want to introduce as little changes to the TCP stack as possible and still get access to all the information needed to model the bandwidth available to a TCP connection.

We implemented our prototype in the NetBSD 1.3 operating system [119]. Other operating systems provide similar support for code that can be dynamically loaded into and unloaded from the operating system kernel. The advantage of using NetBSD is that it has a standard and well-documented TCP stack. (The source code is derived from the original BSD implementation. Wright et al. [199] explain and discuss the entire source code of the popular 4.4 BSD-Lite implementation of TCP.)

TCP stack

For reasons of code maintenance, we keep the changes to the TCP stack to a minimum. In particular, changes to kernel data structures (e.g., TCP control blocks[1]) are completely avoided, since such changes would require to recompile the kernel *and* all applications that rely on these data structures.

We installed a few hooks (call-backs) at appropriate places in the TCP stack, so that asynchronous events such as timeouts etc. can be recorded. When the loadable kernel module is loaded, the hooks are installed and TCP calls back the kernel module to update state information held there (see below). Otherwise, that is, if the LKM is not loaded, TCP operates as if unchanged.

The discussion in Chapter 8 shows that the following information is required to model the throughput of a TCP connection: round-trip time (*rtt*), loss indication probability (p), the average duration of a timeout period (t_{RTO}), the receiver's strategy to generate acks (b), the receiver window size (W_{max}), and possibly the size of the congestion window (*cwnd*). *rtt* is obtained by measurements that are carried out once per round-trip time (in analogy to the *rtt* measurements performed by TCP [199]). Instead of using TCP's coarse-grained measurements, which typically have a granularity of 500 ms, we sample the system clock to obtain accurate estimates with a granularity on the order of μs. For this purpose we install two hooks, one to start/reset a *rtt*-timer and a second to finalize the measurement. To compute p, the number of triple duplicate ack indications (*TD*), the number of timeouts (*TO*), and the number of packets sent (*dataSent*) must be tracked. We install a hook to record when recovery is entered (after receiving the third duplicate ack), and a call-back that is invoked when a retransmission timer fires. The latter hook can also be used to update t_{RTO}. No hooks are required to assess W_{max} and *cwnd*, as this information can be obtained by polling from the LKM (whenever *tcpmond* requests information). The receiver acking policy is difficult to infer dynamically. Most currently used TCP implementations have $b = 2$.

In addition to the four hooks described, we install a call-back to be invoked when a connection's TCP control block is deallocated by the kernel (after the connection terminates). This call-back allows to clean up data structures held in the LKM that are associated with the TCP control block.

tcpmon (loadable kernel module)

The loadable kernel module fulfills three tasks:

System-call API. The module exports the following system calls. Sending applications that want their TCP connections to be monitored must register the socket associated with the connection (*register()*). The application should also unregister the socket after the connection is terminated (*unregister()*). If an application fails to do so, e.g., because it crashes unexpectedly, the call-back installed in the TCP stack ensures that the corresponding data structures are freed eventually.

[1]Each TCP connection is associated with a data structure called "TCP control block". The TCP control block maintains the connection's state. A TCP connection's state includes state variables for data transmission and error recovery, as well as information used by flow and congestion control procedures.

Moreover, the LKM exports two functions that allow *tcpmond* (or any other application) to query the number of connections that are monitored (*getnoftcpmon()*) and the status of all the monitored connections (*getinfo()*). *getinfo()* traverses all TCP control blocks, checks whether the corresponding TCP connection has been registered by the application and returns a list of 4-tuples (IP addresses and port numbers) which identify the monitored connections and a list of performance samples (one per connection). A performance sample for a connection contains the following data: *rtt*, *TD*, *TO*, *dataSent*, t_{RTO}, *cwnd* and W_{max}. All metrics (except for *rtt* and W_{max}) are reported cumulatively to increase resilience against loss of update messages.

Connection management. Since the TCP control block contains no unused data fields that could be used as a pointer to the performance data/connection state that is associated with the connection and held by the LKM, and because we want to avoid changes to kernel data structures, a different type of mapping between a connection's TCP control block and its performance data must be established. We use a simple hashing mechanism where collisions are resolved by chaining. The register function allocates and initializes the necessary data structures and establishes this mapping. The unregister function and the clean-up call-back deallocate the data structures and delete the association with the TCP control block.

Information collection. The information required by TCP throughput models is stored in the LKM. The call-backs installed in the TCP stack invoke functions provided by the LKM that update the connection state held by the LKM. Two call-backs are used for *rtt* measurement. A first function records the system clock when TCP starts or resets the *rtt* measurement for a data segment. A reset can occur if data is retransmitted [88]. A second function is used when TCP terminates the *rtt* measurement. The function again samples the system clock and computes the *rtt* (in μs resolution) as the difference between the two timestamps.

The function called when recovery is entered simply increments the *TD* counter.

The function called when a retransmission timer expires increments the *TO* counter, and checks if the timeout is the first for a particular data segment (i.e., checks that the timer has not been subject to exponential back-off). If timer has not been back-offed, the duration of the timeout period must be computed. There are two options that provide a reasonable tradeoff between accuracy and efficiency. First, the connection's TCP control block contains a variable that reflects the duration of the timeout period in 500ms ticks. The tick-based measurement can deviate from the effective duration of the timeout period by up to 500ms. (Retransmission timers are checked only twice per second). Second, since the fine-grained *rtt* measurement is performed once per round-trip time, a more accurate approximation can be obtained as the period between the last *rtt* measurement and the timeout. This approximation for t_{RTO} is more accurate only if $rtt < 500$ms. The approximation based on fine-grained measurements deviates from the effective timeout period by at most one *rtt*. We currently use the second alternative.

Source file	Original	Changes
netinet/tcp_input.c	2172	33
netinet/tcp_output.c	330	4
netinet/tcp_timer.c	659	9
netinet/tcp_subr.c	750	4
Total	3911	50

Table 9.1: Size of changes to NetBSD 1.3 TCP stack (reported in # lines changed/added). The complete TCP stack comprises 13 files and 5565 lines of code.

Source file	Code	Comment	Total
tcpmon_syscall.h	55	71	126
tcpmon_syscall.c	32	31	63
tcpmon_error.h	28	0	28
tcpmon.h	31	14	45
tcpmon.c	692	213	905
Total	838	329	1167

Table 9.2: Size of loadable kernel module (in # lines).

Bandwidth modelling

tcpmond periodically checks whether any TCP connections exist that are registered to be monitored. If so, it uses the *getinfo()* system call to obtain a list of all monitored connections and their associated performance data. The performance data are encapsulated in update messages that are dispatched to the observer component. *tcpmond*'s task is simple and can therefore be easily integrated with the observer if the observer is co-located with the sending application. The TCP throughput models registered with the observer are then used to model the bandwidth available to each of these connections. Since the throughput models describe asymptotic TCP behavior, the cumulative metrics for the various parameters are used directly (instead of moving some averaging window across the data).

Discussion

One of the questions we want to answer by implementing the different approaches to information collection is how difficult it is to actually implement a transport-level monitor. How many changes to the protocol stack are necessary?

Table 9.1 lists the source files affected by and the size of our changes to the TCP stack. In total, the NetBSD 1.3 TCP stack comprises 13 files and a total of 5565 lines of code. The table shows that only 50 lines of code in 4 files had to be changed or added to install the call-backs mentioned above. In other words, the changes affect less than 1% of the total TCP stack.

Table 9.2 lists the files that comprise the loadable kernel module. The first two files define the programming interface to be used by (and linked to) applications such as *tcpmond*. The core of the LKM is the file *tcpmon.c*, which implements the functionality described above. The table illustrates that the LKM is a fairly small piece of software with less than 900 lines of code. A more detailed breakdown of the 692 lines of code for *tcpmon.c* shows that about one third each is used to implement the following three tasks: LKM administration/system call dispatching,

connection management, and the implementation of the system calls. The implementation of the call-back functions is lean and uses only about 60 lines of code.

From these observations, we conclude that the kernel changes required to make transport-level monitoring work and to achieve the proposed widening of the transport API are indeed very small. This finding and the results of Chapter 8 indicate that an instrumentation of the TCP stack to allow for transport-level monitoring may be a worthwhile addition to any TCP stack as it can be easily accomplished. The clean distinction between the hooks in the TCP stack and the implementation of the call-back functions in the LKM turned out to facilitate experimentation with and maintenance of the code.

9.3 Evaluation

9.3.1 Methodology

In this section we quantitatively compare the monitoring approaches with respect to the application-relevant criteria identified in Section 7.3: efficiency of the information collection process, and the quality of the bandwidth information.

We distinguish two issues for efficiency. First, bandwidth monitoring should impose as little overhead on the network and on the end-system(s) as possible. Second, monitoring should be scalable so that a large number of simultaneous connections can be handled without incurring significant overheads that distort network and application performance. We expect monitoring overhead to be dependent on the method of information collection, the (targeted) sampling frequency, and the number of connections that must simultaneously be monitored.

There are two aspects that determine the quality of bandwidth estimates. First, the bandwidth estimates provided to the application must accurately reflect the available bandwidth. Second, these estimates must be provided in a timely fashion, that is, a bandwidth monitor must detect and report changes in available bandwidth quickly. Timeliness depends on the sampling frequency with which performance information is obtained and the averaging window used to compute the estimates. The (effective) sampling frequency is limited by the overheads incurred by monitoring. The issue of accuracy has been addressed in Chapter 8 and is not further discussed here. Chapter 8 showed that transport-level monitoring is able to accurately model the bandwidth available to an application. Furthermore, the bandwidth models capture the behavior of long-running connections, and we expect that these models may therefore be better suited for prediction of bandwidth availability in the future than application-level bandwidth estimates.

We conduct a simple experiment to study how the different approaches to information collection behave under load, since both issues, efficiency and timeliness, seem to depend on how well the monitoring system can cope with load.

Experimental setup

A 200 MHz Pentium Pro PC with 128 MB RAM running NetBSD 1.3 (with our TCP modifications) acts as the server machine. A dual-processor 300 MHz SPARC Ultra 4 system with 1 GB RAM running SunOS 5.6 serves as the client machine. The two hosts are connected with a 100 Mbit/s switched Ethernet. *observd* is co-located with the server, that is, runs on the PC.

As discussed above, we expect the load on the monitoring system to be determined by the number of connections n that must be monitored simultaneously and the frequency f with which performance reports should be generated. We run the following experiment. The client machine starts n client processes simultaneously. Each client process connects to the server and requests transmission of $100/n$ MB. The client requests specify the method for monitoring (*app-rcv*, *app-snd*, or *tcpmon*) and the sampling frequency f. The appropriate monitoring components are instantiated after connection establishment. For *tcpmond* we explore two variants. In the first alternative, *tcpmond* is an independent component on the server host that communicates with *observd* by means of UDP messages (we refer to *tcpmond*'s situation as "separated"). With the second variant, *tcpmond* is "integrated" with *observd*, which implies that update messages from *tcpmond* are passed to *observd* by means of procedure calls. In contrast, the *app-rcv* and *app-snd* stubs are part of the client and server application, respectively, and communicate with *observd* by means of UDP messages.

The sampling frequency f determines how often the monitoring component of choice must generate performance reports. For application-level monitoring we compare two alternatives for the generation of performance reports. First, a performance report is generated (by the application stub) whenever application data is sent/received (i.e., whenever the respective socket function returns) *and* $\delta t = 1/f$ seconds have passed since the last report has been sent. Note that this mode of operation adapts to the transmission rate of application data. If no application data is sent/received, no performance updates are produced. We term this alternative "adaptive". Second, the application stub can generate performance reports at the rate f, regardless of the progress of application data transfer. We refer to this mode of operation as "strict" feedback generation.

We vary the sampling interval $\delta t = 1/f$ between 0.05 and 2.0 seconds, and choose the number of parallel connections n from the set $\{5, 10, 20, 40\}$. We run 10 experiments for each combination of n, δt, and the method for information collection: *app-rcv* (strict and adaptive), *app-snd* (strict and adaptive), and *tcpmon* (separated and integrated). Unless otherwise noted, the results reported in the following sections are specified in terms of mean and a confidence interval for the mean that is computed at a confidence level of 95%.

9.3.2 Efficiency

The end-system overhead incurred, that is, the CPU bandwidth consumed, by monitoring depends on two issues: the overhead incurred to obtain and process a single performance sample, and the number of samples that must be processed by the observer per second. In principle, the number of samples that must be collected and processed each second is equal for all the monitoring techniques and is given by $n \cdot f$. Thus, we expect the end-system overheads to be mainly effected by the costs incurred by a single performance sample.

Cost per sample

Figures 9.3 (a)–(d) show the overhead incurred by a sample as a function of the monitoring technique, the sampling interval δt, and the number of connections n. Two types of costs are reported. The total per-sample costs of information collection (solid line) and the processing costs for a sample (dotted line). The processing costs are included in the total costs. The costs

reflect the user and system CPU time consumed[2]. The total per-sample costs are computed by dividing the CPU consumption of *observd* for the whole experiment by the number of all the performance samples that are processed. The per-sample processing costs are measured for each performance sample individually and cover the time needed to store and process the information conveyed in the update message. Hence, the processing costs do not cover the time required to receive and identify a performance sample.

There are three aspects to note in Figures 9.3 (a)–(d). First, *tcpmon* (Figures (c) and (d)) incurs considerably lower total costs per sample than the application-level approaches (Figures (a) and (b)). In addition, *tcpmon*'s processing costs per sample are also smaller than those of the other approaches. The discrepancy in terms of total costs between *tcpmon* and the other approaches can mainly be attributed to the fact that *tcpmon* incurs considerably smaller communication overheads: the performance samples of all n connections are "batched" in *tcpmon*. That is, the performance samples of all n connections are aggregated and communicated to *observd* in a single update message. This is in contrast to *app-snd* and *app-rcv* where each performance report must be processed individually (because they are generated by different applications). The effect of this "batching" is reflected by the fact that the total per sample costs decrease as the number of connections increases (Figures (c) and (d)). This observation implies that the constant cost of generating and communicating such a batched performance sample is amortized over larger numbers of connections.

Second, the total per-sample costs seem to depend on the sampling frequency. This tendency is most pronounced for the application-level approaches (Figures (a) and (b)). The higher the sampling frequency (i.e., the smaller the sampling interval δt), the smaller are the per-sample costs. This correlation can be explained by constant *observd* overheads, e.g., start-up overheads, that are amortized over larger numbers of samples. The correlation may also be effected by caching. The higher the number of samples to be processed per second, the more likely it is that instruction and data caches are valid.

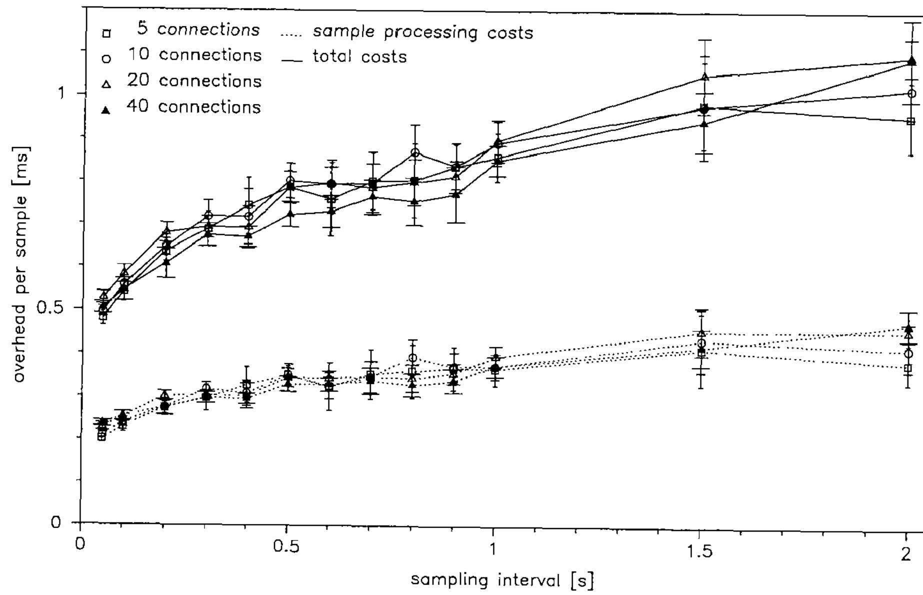
Third, the difference between total costs and processing costs per sample for the application-level approaches (Figures (a) and (b)) is quite significant. The difference may in part be attributed to constant overheads that are included in the total costs, but are not included in the processing costs. This conclusion is supported by the observation that the difference becomes smaller as the sampling interval increases. In addition, the difference is explained by the costs for receiving and identifying a message as a performance report. A further source of overhead not included in the processing costs are format conversions (from an end-system independent representation) of the performance data. Even though we tried to minimize such sources of overhead there may be still room for optimizations.

Furthermore, note that the small difference between total costs and processing costs for the scenario with integrated *tcpmon* (Figure (d)) indicates that information collection in the TCP stack and the loadable kernel module is very efficient and incurs only very small overheads.

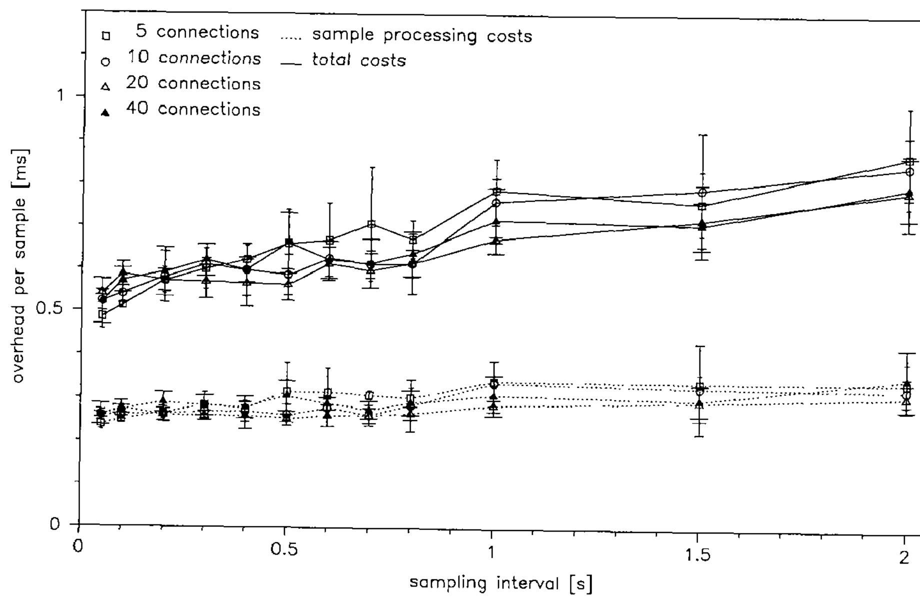
These results are summarized in the first two rows of Table 9.3. The averages reported are taken over all the experiments conducted, i.e., over all combinations of n and δt.

---

[2]NetBSD's accounting of the CPU resources consumed by a process is trustworthy—even for small time scales—since NetBSD accurately measures the time a process is running. This accurate accounting method stands in contrast to other accounting schemes that rely on statistical sampling [114].

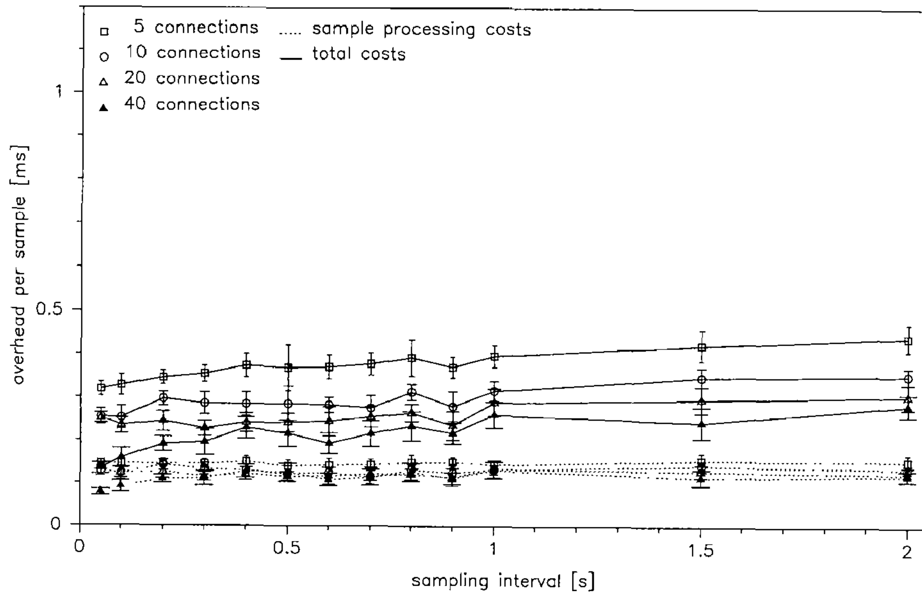


(a) *app-rcv*

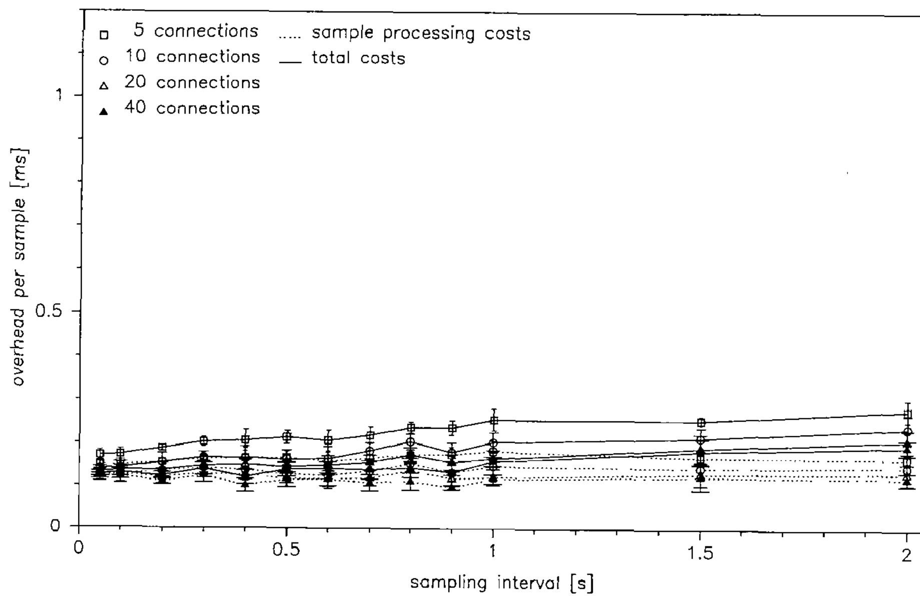


(b) *app-snd*

Figure 9.3: Cost per sample as a function of sampling interval ($\delta t = 1/f$) and number of connections n.



(c) *tcpmon, separated*



(d) *tcpmon, integrated*

Figure 9.3: Cost per sample as a function of sampling interval ($\delta t = 1/f$) and number of connections n.

Total overhead

As a consequence of the observations on the per-sample costs we would expect *tcpmon* to incur significantly smaller monitoring overheads than the application-level solutions. We express monitoring overhead as the average CPU time consumed by *observd* each second. Figures 9.4 (a)–(f) plot the monitoring overhead per second as a function of δt and n. The figures report both the measured overhead (solid line) and the expected overhead (dotted line). The expected overhead is based on the measured per-sample costs and the number of samples that are expected to arrive at the observer per second ($n \cdot f$).

Surprisingly, the measured overheads per second seem to be comparable for all monitoring approaches (all approaches consume less than 10% of the CPU bandwidth). Furthermore, application-level, receiver-based monitoring with adaptive feedback generation (Figure (a)) seems to incur the smallest overheads when compared with transport-level monitoring and the other variants of application-level monitoring. This observation is counter-intuitive when taking into account the findings of the previous section, which reported the highest per-sample costs for the configuration *app-rcv, adaptive*. The average monitoring overheads for the different approaches are summarized in the third row of Table 9.3.

We note that the measured overhead for application-level monitoring (Figures (a)–(d)) deviates quite considerably from the expected overhead. In contrast, transport-level monitoring (Figures (e) and (f)) matches the expectations quite well (except for $n = 40$ and $\delta t < 0.3$). Among the application-level approaches, adaptive feedback generation exhibits larger deviations than strict feedback generation. Likewise, we note that sender-based monitoring better matches the expected overheads than receiver-based monitoring.

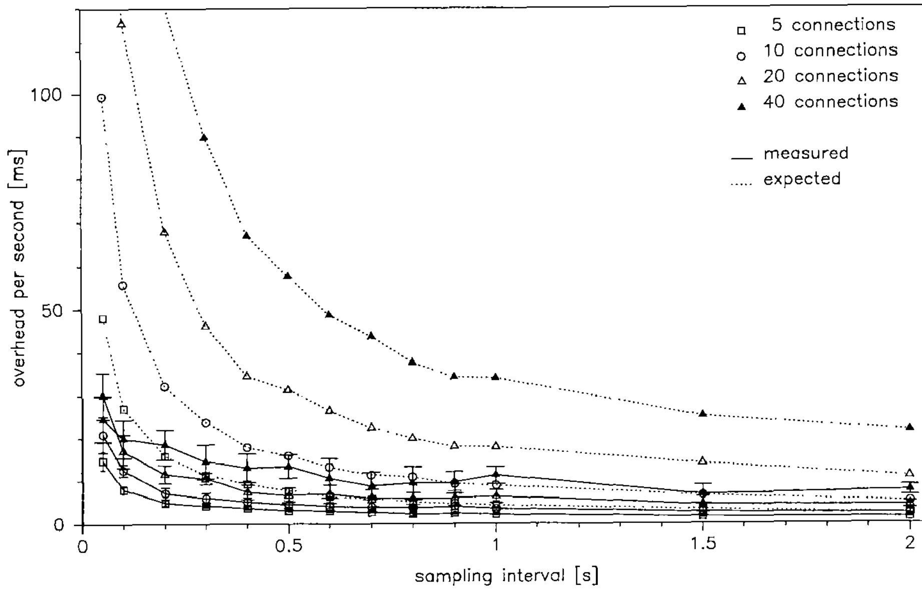
Aggregate sampling rate

These discrepancies between measured and expected overheads indicate that—contrary to our initial belief—the total number of samples received per second (by *observd*), that is, the aggregate sampling rate, varies significantly between the different approaches to information collection.

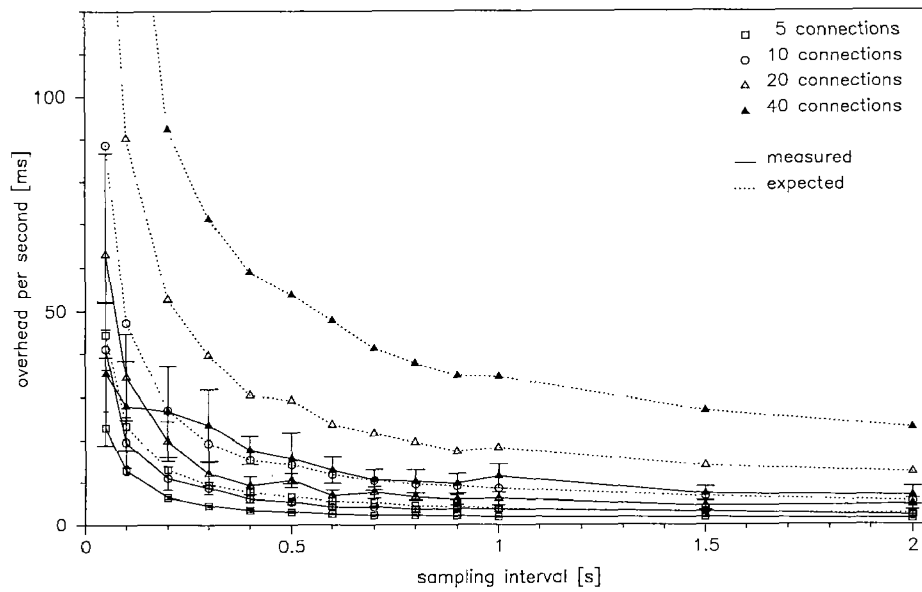
There are mainly two reasons for these deviations. First, fewer samples than expected are generated by the sensors each second. Second, some of the generated samples are dropped by the network or the end-systems.

The row termed "generated" in Table 9.3 reports the fraction of the $n \cdot f$ samples that are actually generated. On average, only about one third of the expected sampling rate is achieved by the approaches with adaptive feedback generation. Strict feedback generation achieves approximately 60% of the expected rate, whereas transport-level monitoring achieves around 94%. In addition, we find that for application-level monitoring, the fraction of samples generated is primarily effected by n, the number of simultaneously active connections.

The row termed "lost" in Table 9.3 reports the fraction of the generated samples that are dropped by the network or the end-systems. Only the application-level, receiver-based mechanisms are affected by lost samples (as these are the only mechanisms that incur network overhead). Thus, we conclude that samples are primarily dropped by the network. The loss rate is 24% for adaptive and 34% for strict feedback generation. Furthermore, we note that the loss rate is largely unaffected by f and n.

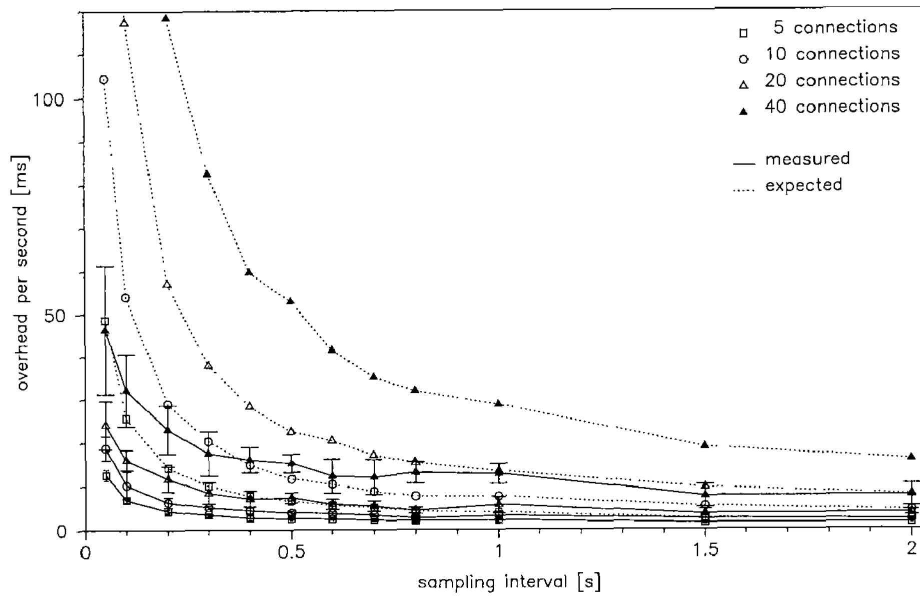


(a) *app-rcv, adaptive*

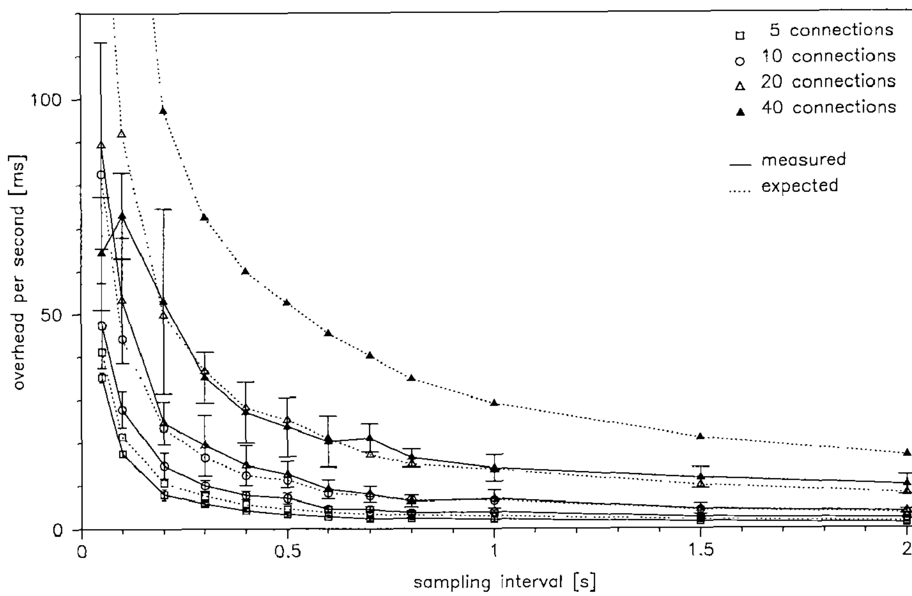


(b) *app-rcv, strict*

Figure 9.4: Monitoring overhead (CPU time in ms consumed) per second.

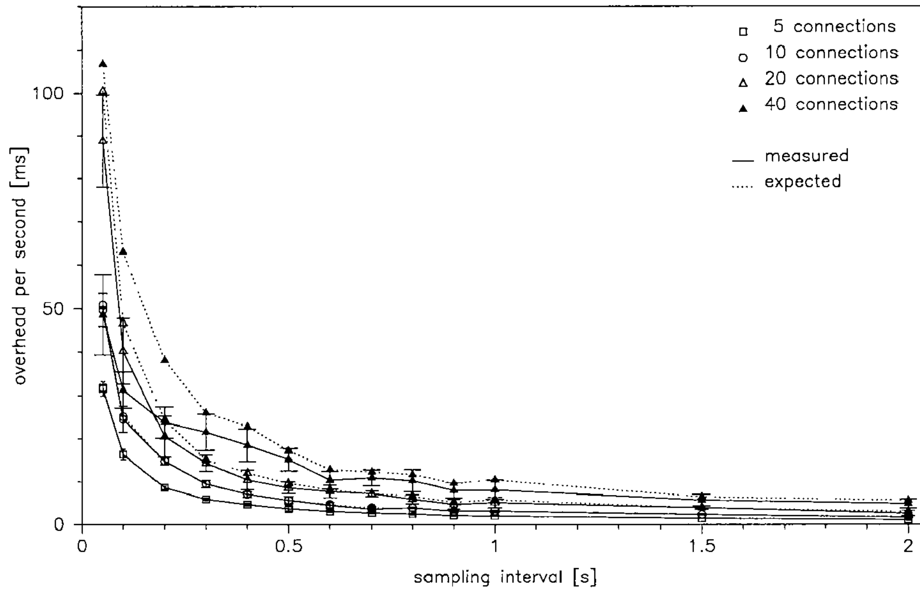


(c) *app-snd, adaptive*

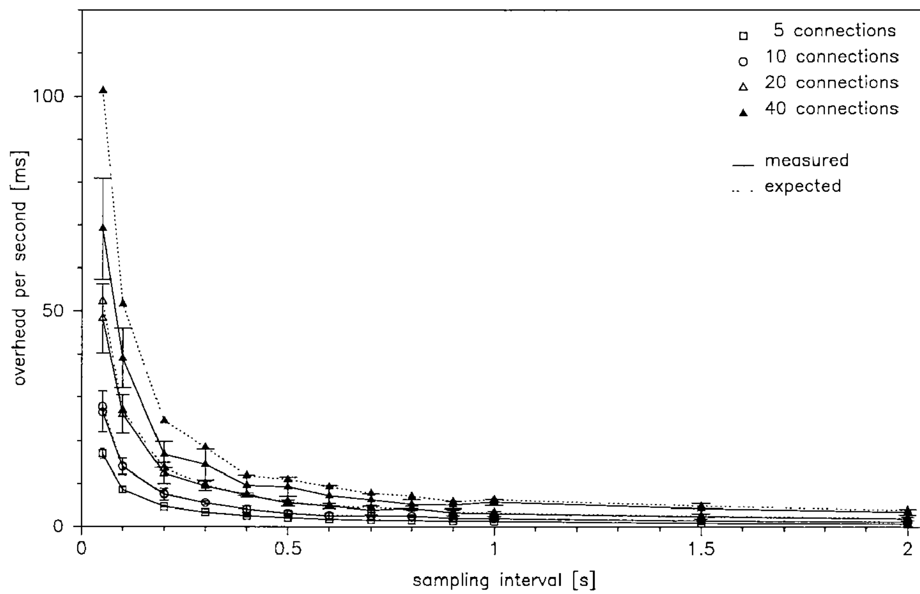


(d) *app-snd, strict*

Figure 9.4: Monitoring overhead (CPU time in ms consumed) per second.



(e) *tcpmon, separated*



(f) *tcpmon, integrated*

Figure 9.4: Monitoring overhead (CPU time in ms consumed) per second.

The observations on the number of performance samples that are actually generated imply that application-level monitoring is highly dependent on the network and end-system load incurred by applications (and thus may be effected by the choice of experimental setup).

Discussion

The fact that many of the performance samples are not generated or lost with application-level monitoring has two consequences. First, the quality of the bandwidth information, in particular, the timeliness of the information is negatively affected (as is shown in Section 9.3.3). Second, the conclusion that application-level monitoring is more efficient than transport-level monitoring—a conclusion that could be drawn from the measured overheads reported in Figures 9.4 (a)–(f)—must be qualified. With a different and more realistic experimental setup, the application load on the server and client machines would be smaller and thus the number of performance samples that are generated in scenarios with application-level monitoring would be considerably higher. As a consequence, the overheads incurred for these approaches would be higher and would more closely match the expected overheads (and thus exceed the overheads of transport-level monitoring).

A more realistic scenario would have n different client machines connect to the server. In such a configuration, the client hosts are less loaded, and thus, the *app-rcv* sensors can produce the samples at the targeted sampling frequency f. (For strict feedback generation this setup ensures that 100% of the samples can be generated; instead of only 60% as in our experiments.) For sender-based monitoring the application load on the server host is the decisive aspect. In a more realistic scenario the server distributes its load to other (back-end) servers [137]. Such a configuration reduces the application load on the server(s)—with the effect that the number of samples generated each second more closely matches the targeted aggregate sampling rate. As a result, the overheads incurred by application-level would be considerably higher than measured in our simplistic experiments.

Impact on application performance

So far, we only discussed the monitoring overheads incurred by *observd*. The second efficiency concern listed in Section 9.3 states that monitoring should be able to handle a large number of simultaneous connections without incurring overheads that distort network and application performance. We briefly comment on the impact of the monitoring methods on application performance. The row termed “duration” in Table 9.3 reports the average duration of all the experiments conducted with a particular monitoring technique. Duration is a valid metric for comparison, because in each experiment a constant amount of application data (100 MB) is transferred from server to clients (each client requests $100/n$ MB). If no monitoring is performed the average duration of the experiments is 28.6 seconds. The confidence interval for the mean is $[27.7, 29.5]$. We find that experiments with application-level, receiver-based or with transport-level monitoring take at most 10% longer to complete. However, application-level, sender-based monitoring affects application performance significantly. Compared to the other monitoring approaches, the experiment duration is approximately 50% higher for sender-based, adaptive and about 100% higher for sender-based, strict feedback generation.

	app-rcv				*app-snd*				*tcpmon*			
	adaptive		strict		adaptive		strict		separated		integrated	
Metric	μ	e_{95}	μ	e_{95}	μ	e_{95}	μ	e_{95}	μ	e_{95}	μ	e_{95}
Processing cost [μs]	330	7	315	8	286	5	262	6	128	2	135	3
Total cost [μs]	759	16	725	19	639	13	570	15	284	7	174	4
Overhead [ms]	8.5	0.7	9.9	1.1	9.0	1.2	17.3	2.6	12.7	1.4	9.4	1.3
Generated [%]	39.8	1.7	58.8	1.7	33.4	1.7	59.1	1.8	94.2	1.2	94.4	1.3
Lost [%]	23.6	0.5	34.2	0.8	0.0	0.0	1.9	1.2	0.0	0.0	0.0	0.0
Duration [s]	30.5	0.9	29.4	0.9	43.3	1.3	69.5	2.6	31.1	0.8	31.2	0.8
Timeliness[a]	1.4	—	1.3	—	1.4	—	1.0	—	1.0	—	1.0	—
Variability[b]	0.9	5.1	0.9	3.6	0.9	4.4	0.9	1.7	1.0	1.1	1.0	1.0

[a] μ is the median deviation from the sampling frequency defined as *sample interarrival time/sampling interval*.
[b] μ and e_{95} reflect the 10- and 90-percentile of the deviation from the sampling frequency, respectively.

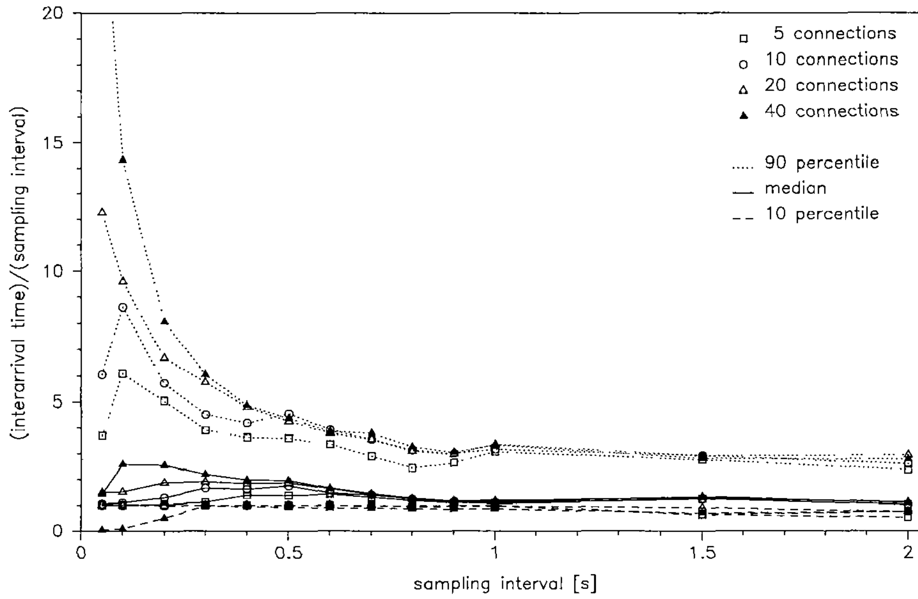
Table 9.3: Summary of comparison. Averages (μ) are computed over all experiments conducted, that is, over all combinations of n and δt. The columns e_{95} report the size of the two-sided confidence interval for the mean at a confidence level of 95%. The confidence interval for the mean μ is given as $[\mu - e_{95}, \mu + e_{95}]$.

9.3.3 Timeliness

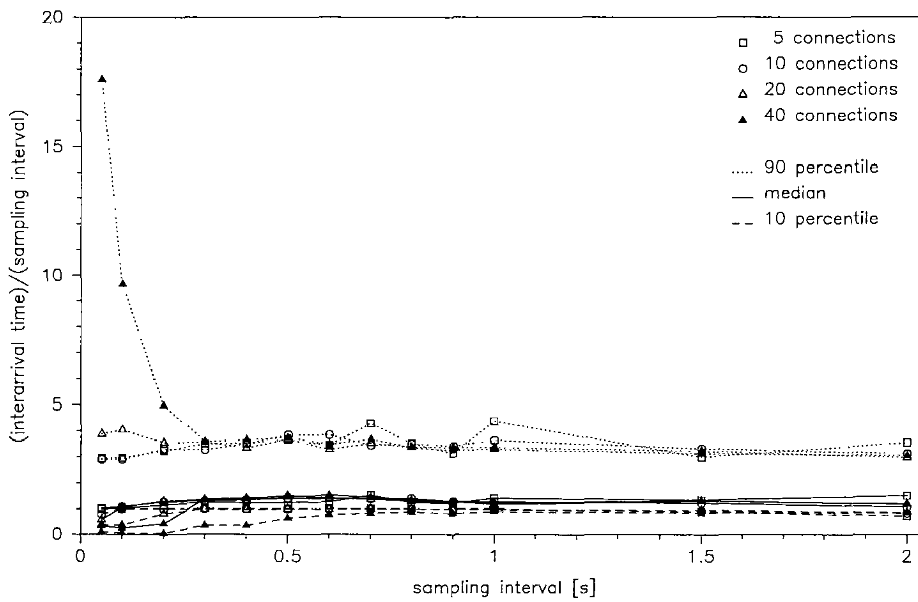
Bandwidth estimates must be provided in a timely fashion, that is, a bandwidth monitor must detect and report changes in available bandwidth quickly. Timeliness primarily depends on the sampling frequency with which performance information is obtained[3]. As shown in the previous section, the (effective) sampling frequency is limited by the monitoring overheads (and the application load). Up to now we have only studied the average number of performance samples that arrive at and are processed by *observd* each second. An important aspect of timeliness that has not been addressed so far is *when* the samples arrive, that is, whether they are regularly spaced or not. If k performance samples arrive per second and all the k performance samples arrive in a time frame $\delta t \ll 1$ second, then all samples except for the most recent sample are useless for the application and simply waste end-system and network resources. We use the interarrival time between successive performance samples for a connection to describe the regularity of sample arrivals at the observer. We compute the distribution of interarrival times for an experiment and store the median, as well as the 10- and the 90-percentiles of the distribution. The values reported are the 10-, 50-, and 90-percentile divided by the (targeted) sampling interval δt. A ratio of 1 means that the samples are spaced exactly with the targeted sampling interval δt.

Figures 9.5 (a)–(f) plot these ratios as a function of monitoring approach, sampling interval, and number of connections. We find that application-level monitoring with adaptive feedback generation (Figures (a) and (c)) witnesses considerable deviations from the targeted sampling intervals. The median interarrival time is up to a factor three larger than intended for large numbers of connections and high sampling frequencies. The other approaches have median interarrival times that match δt quite accurately. The application-level approaches experience considerable variation in the interarrival times as can be seen from the 10- and 90-percentiles.

[3] The timeliness also depends on the averaging window used to compute the estimates. Since the averaging window can be chosen (almost arbitrarily) by the application, we exclude this factor for the comparison.

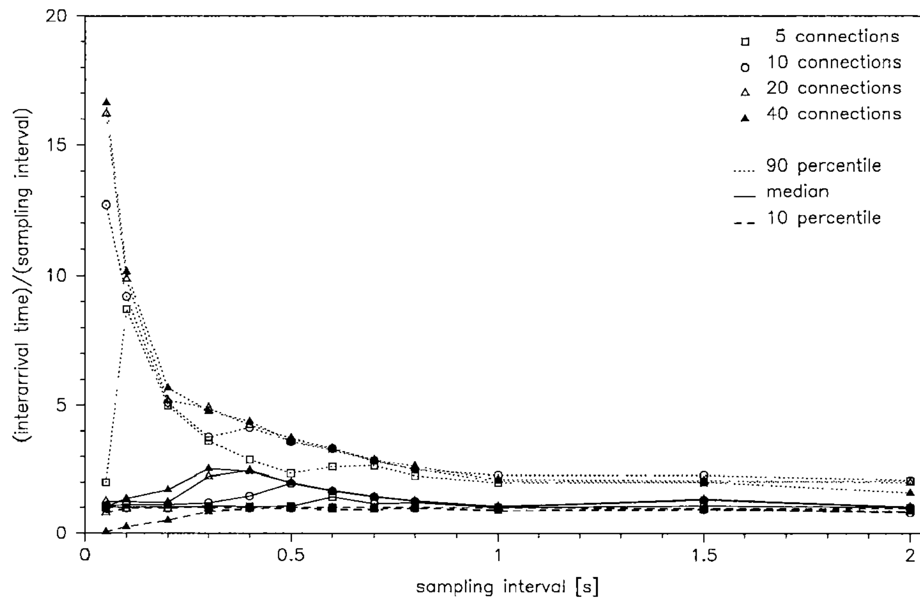


(a) *app-rcv, adaptive*

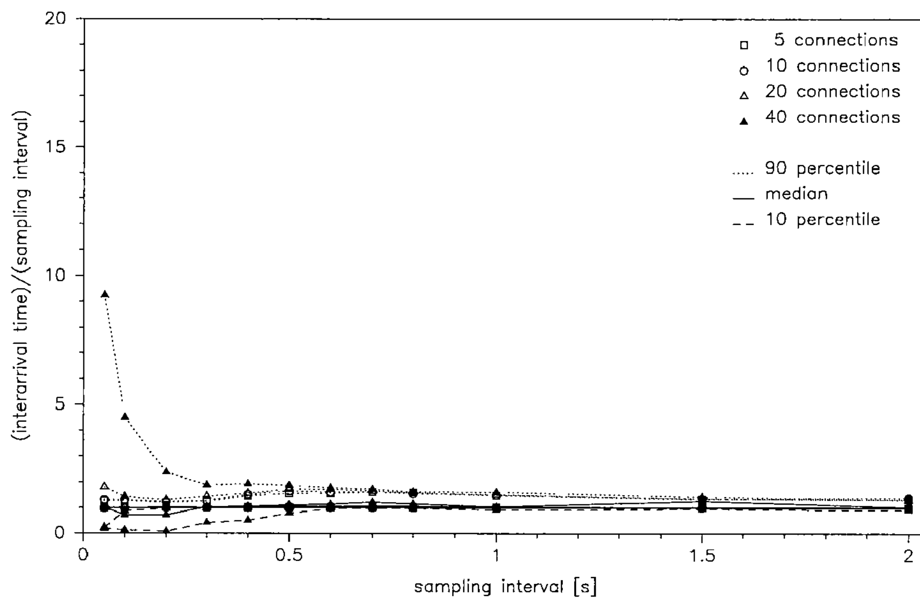


(b) *app-rcv, strict*

Figure 9.5: Timeliness of the bandwidth information.

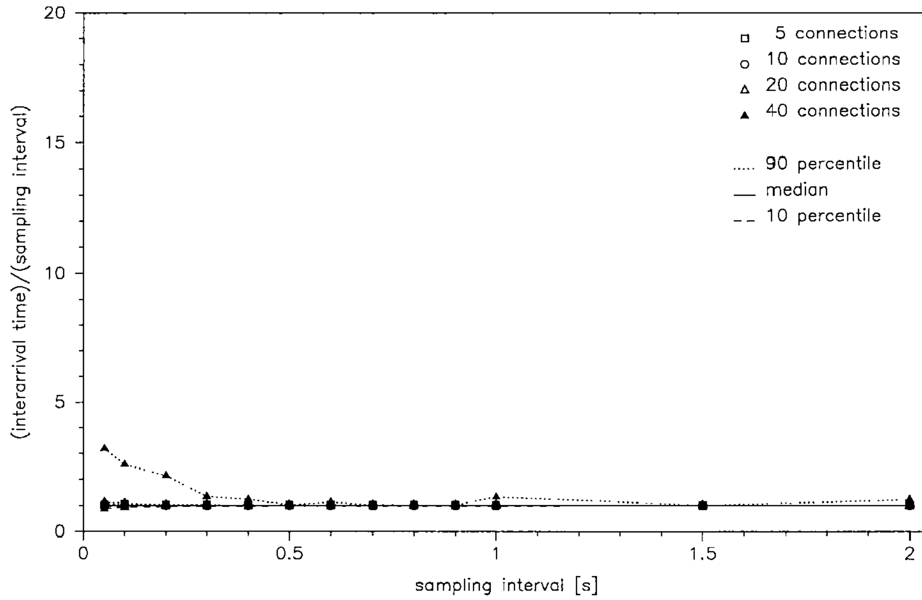


(c) *app-snd, adaptive*

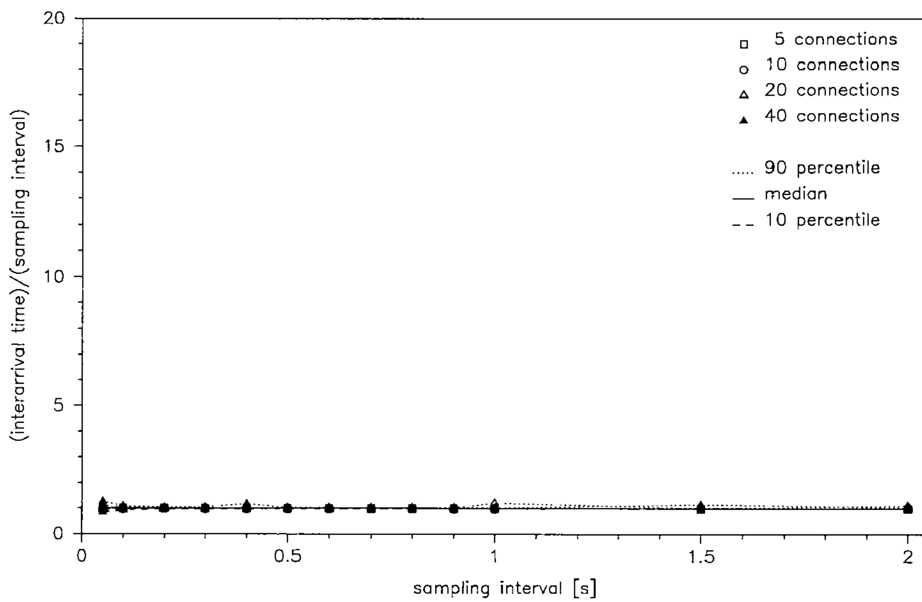


(d) *app-snd, strict*

Figure 9.5: Timeliness of the bandwidth information.



(e) *tcpmon, separated*



(f) *tcpmon, integrated*

Figure 9.5: Timeliness of the bandwidth information.

Of all application-level approaches, only sender-based monitoring with strict feedback generation matches the targeted sampling intervals quite closely. In contrast, transport-level monitoring very accurately matches the targeted sampling intervals for all combinations of n and f. Furthermore, the variability is negligibly small. The timeliness of the bandwidth information increases steadily from Figure (a) through Figure (f). These findings are summarized in the last two rows of Table 9.3.

9.4 Summary

This chapter presents an architecture for on-line bandwidth monitoring that integrates and unifies different techniques of information collection about network status. The monitoring system allows multiple monitoring approaches to be used simultaneously. The architecture is extensible with respect to new techniques for information collection, bandwidth modelling, and bandwidth prediction. Furthermore, we demonstrate that transport-level monitoring can be implemented easily, requiring only minimal changes to the TCP stack.

The comparative evaluation of different monitoring approaches shows that transport-level monitoring incurs small costs (per performance sample). The costs are significantly smaller than the per-sample costs incurred by application-level approaches. In terms of overall overhead the approaches seem to perform comparably. However, we argued in Section 9.3.2 that this result is a consequence of our simplistic (and not very realistic) experimental setup. We claimed that in more realistic scenarios the overheads witnessed by application-level approaches exceed those of transport-level monitoring. Moreover, we find that transport-level monitoring does not distort application performance. (The same applies to application-level, receiver-based monitoring.) In addition, transport-level monitoring is capable of producing a regularly spaced stream of performance samples. In contrast, application-level approaches exhibit highly fluctuating sample interarrival times that often exceed the targeted sampling intervals considerably.

To conclude, Chapters 7–9 show that information about network resource availability can be provided both accurately and efficiently by light-weight monitoring at the transport-level. Our experience with a prototype monitoring system demonstrates that such a monitoring scheme requires only minimal changes to existing protocol stacks and should be simple to incorporate in the design of new transport protocols. Our findings also imply that a simple widening of the application programming interface should suffice to provide network-aware applications with the information about network status sought.

Seite Leer /
Blank leaf

Chapter 10

Conclusions

Besides providing the "right" services and information, a key factor to the success of an Internet content provider is whether the provider is able to serve its users at predictable and satisfactory levels of quality and performance. An important criterion for quality is that the content is delivered to the user in a timely manner. Ideally, the user should be able to specify how long she is willing to wait for the data and how she values the quality of the data delivered. Thus, content providers must be "smart" about how they satisfy individual user requests: they must be able to deliver as much high-quality data that is relevant to the user's information needs as possible within the time frame allotted.

There are two problems that hamper the provision of such a service model for networked applications in today's Internet: heterogeneity (in client capabilities and bandwidth) and fluctuations in available bandwidth. The fact that bandwidth supply can differ and fluctuate significantly results in highly unpredictable application behavior, which is often intolerable from a user's point of view. *Network-aware applications* provide a solution to these problems: they dynamically adapt their resource demands to match the varying supply of bandwidth with the goal to achieve predictable response times. Such *network-aware* applications must often trade network resources for some *measure of quality* of the data delivered.

Although a considerable body of related research exists in the field of network-aware applications, a number of challenges remained largely unaddressed by these previous studies (see Chapter 1). For instance, many adaptive applications make only static adaptation decisions with the goal to accelerate content delivery for clients with low-speed network access, but do not take user preferences on response time or quality into account. Furthermore, although adaptive applications are quite complex software systems, the process of building adaptive applications has barely received attention and many current network-aware applications are built in a rather ad-hoc manner and are often tailored to the application's specific needs. In addition, network-aware applications exhibit fairly complex dynamic behavior which is difficult to evaluate thoroughly.

10.1 Contributions

This dissertation addresses the challenges mentioned and makes the following contributions:

Framework. The dissertation presents a reusable framework for network-aware applications. In contrast to other work, the issues of *quality-* and *system-awareness* are an integral part of the framework and the applications built thereon. Quality-awareness means that applications try to be smart about how to reduce the quality of the data delivered, so that the user-specified time limit can be met, and so that the negative impact on the overall service quality is minimized. The framework is flexible and can deal with a wide range of definitions of quality. The adaptation framework accounts for other resources (e.g., CPU) that are employed to make the response time-quality tradeoff (system-awareness).

The framework's sender-initiated, model-based adaptation supports such an integral approach to service quality and lends itself well to fast deployment. The dissertation shows that the framework-based approach to the development of such network-aware applications allows for reuse of the core adaptation (i.e., decision making) process and can therefore shield developers from many of the complexities in dealing with network dynamics.

Application dynamics. The dissertation presents a systematic approach to the evaluation of the complex dynamic behavior of adaptive applications which goes beyond many of the ad-hoc evaluations carried out in previous work. The evaluation establishes that model-based adaptation is robust with respect to many of the parameters that influence adaptation decisions. Adaptation performance, defined as the application's ability to meet a user-specified time limit, is not affected regardless of how many of the factors influencing adaptation decisions vary. Performance considerations boil down to the communication-computation tradeoff striven for by network-aware applications. We find that network-aware content delivery can provide predictable response times over a wide range of bandwidths and CPU powers.

Compared to a static delivery policy, adaptation can provide a benefit to the user because it is able to deliver the response within a user-specified time limit and because it can be smart about which of the objects delivered must be reduced in quality to attain the goal on time and to maximize the utility of the response. Furthermore, we note that adaptation incurs only small overheads.

Application-network coupling. To the best of our knowledge, this dissertation is the first study that quantifies the impact of the quality of information (about bandwidth availability) on the performance of network-aware applications. The dissertation shows that the performance of network-aware applications, that is, the ability to meet a user-specified time limit and to achieve high bandwidth utilization, depends strongly on the accuracy and timeliness of information about network resource availability.

Furthermore, the dissertation demonstrates that the need for accurate and timely information about network resource availability can both effectively and efficiently be satisfied with transport-level monitoring. Transport-level monitoring compares favorably with application-level monitoring as far as the timeliness of the bandwidth estimates and the overhead incurred by monitoring are concerned. In addition, our evaluation indicates that the ability to predict bandwidth depends considerably on the transport protocol used. Our prototype monitoring system demonstrates that the implementation of such a monitoring scheme requires only minimal changes to existing protocol stacks and a simple widening of the application programming interface.

10.2 Future work

There are three major directions for future research based on our work. The adaptation mechanisms and the abstractions provided by the framework can be refined, resource prediction can be improved, and a more integral solution to the start-up problem can be adopted.

Reusable software structures. The framework for network-aware applications can be improved in three ways. First, the adaptation mechanisms can be refined, e.g., to incorporate client processing speed, to minimize the impact of communication idle times, or to deal with the varying accuracy of resource predictions (see Section 4.9). Furthermore, an evaluation of the mechanisms proposed and implemented to improve the start-up behavior (Section 4.7) and application agility (Section 4.8) would yield interesting feedback that could help to refine the framework's adaptation mechanisms.

Second, the framework can be extended in several dimensions (see Section 5.2). The framework can be extended for other Chariot-like applications with respect to the transformation algorithms used, as well as the encodings and media types supported. The framework can be adapted for use in network-aware Web object delivery, or it can be retargeted with minor adjustments to serve other, slightly different, application domains, e.g., to meet a reservation or a cost budget. We expect that the implementation of such framework extensions provides valuable feedback to the design of the framework. Such feedback would allow to refine the abstractions provided by the framework and hence improve framework reusability for prospective developers of network-aware applications.

Third, one could study whether and how the framework could be adapted to different application domains that have been explicitly excluded from the discussion so far (see Sections 3.1 and 5.2.4). For example, future work could investigate how the framework could be integrated with nodes providing active services within the network [6], e.g., to support network-aware content delivery in a multicast scenario.

Resource prediction. Although application performance is not overly sensitive to the accuracy of the resource models the dissertation has provided empirical evidence demonstrating that inaccurate information about resource availability can have a significant negative impact on performance. While we have shown how information about network status can be gathered efficiently, more work is needed to understand whether and how the bandwidth available to an individual application can be predicted. How far into the future can we predict bandwidth availability? Which prediction models are best suited to capture the bandwidth fluctuations experienced in today's networks? The investigation by Sang et al. [156] may provide a starting point for these issues.

In addition, our framework would also benefit from more sophisticated prediction mechanisms that can estimate the CPU resources available to a network-aware application. The analysis of host load prediction models conducted by Dinda et al. [41, 42] may serve as a start.

Start-up. An important issue for the type of network-aware applications considered in this dissertation is the behavior at application start. Start-up behavior is a recurring problem on at least three different (but related) levels. First, how does a user know in advance how to set a

realistic time limit (Section 4.9)? Second, how should a network-aware application operate if no bandwidth estimates are available (Section 4.7)? And third, how to get reliable bandwidth estimates in the early stages of a TCP connection (Section 8.6)? The three questions are related because they can all be mitigated by resource discovery mechanisms such as those described in Section 7.2.1. Thus, future research in the area of network-aware applications should take resource discovery techniques (bandwidth probing and caching) into account, and should try to integrate network-aware applications with systems that provide or make use of such discovery techniques (e.g., [174, 145, 62]).

10.3 Concluding remarks

Due to their complex dynamic behavior, network-aware applications are both notoriously difficult to construct and to evaluate. Our experience with a framework-based approach to the development of network-aware applications suggests that the complex adaptation decision making process employed by modern adaptive applications can well be encapsulated by an application-level framework. Capturing the essentials of the control flow of such software feedback systems, a framework can shield developers from the run-time complexity of adaptive applications and allow the developers of new applications to focus on their application domain, that is, the data/media types, transformation algorithms, etc. to be supported by the application.

As our work has shown, adaptive applications are dependent on accurate and timely feedback about resource availability (both in terms of network and end-system resources). Our experience with a transport-level monitor allows us to draw conclusions that tie in closely with those purported by Noble [130]. We find that a collaboration between the system and the applications is well suited to support network-aware applications: applications must have a say on how to adapt; on the other hand, the system can more accurately and efficiently measure resource availability.

The second major theme of our work is the importance of a careful evaluation. Identifying the factors that effect application performance and subjecting the network-aware application to varying levels of these factors in a controlled, simple and repeatable fashion enables a thorough performance analysis. Without such an analysis, only few convincing quantitative conclusions could be reached as far as performance and robustness of adaptive applications are concerned.

Network-aware content delivery is a promising technique to provide predictable service quality to the user in today's heterogeneous and volatile network environments. Although this dissertation makes a number of contributions towards a better understanding of network-aware applications, a few problems remain open. Our framework for network-aware applications can serve well as a base from which new applications can be built and new directions in network-aware content delivery and network-aware computing can be explored with reasonable efforts.

Bibliography

[1] A. Adl-Tabatabai, T. Gross, and G.-Y. Lueh. Experience with code reuse in an optimizing compiler. In *Proceedings of the International ACM Conference on Object-Oriented Programming Languages and Applications (OOPSLA)*, pages 51–68, October 1996.

[2] A. Albanese, J. Blömer, J. Edmonds, M. Luby, and M. Sudan. Priority encoding transmission. *IEEE Symposium on Foundations of Computer Science*, pages 604–612, November 1994.

[3] M. Allman and V. Paxson. On estimating end-to-end network path properties. In *Proceedings of ACM SIGCOMM '99*, pages 263–274, August 1999.

[4] M. Allman, V. Paxson, and W. Stevens. TCP congestion control. Request for Comments (RFC 2581), Internet Engineering Task Force, April 1999.

[5] P. D. Amer, S. Iren, G. E. Sezen, P. T. Conrad, M. Taube, and A. Caro. Network-conscious GIF image transmission over the Internet. *Computer Networks*, 31(7):693–708, April 1999.

[6] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178–189, August 1998.

[7] Apache HTTP server. `http://www.apache.org`.

[8] AvantGo. `http://www.avantgo.com`.

[9] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *Proceedings of IEEE Infocom '98*, March 1998.

[10] H. Balakrishnan, M. Stemm, S. Seshan, and R. H. Katz. Analyzing stability in wide-area network performance. In *Proceedings of ACM SIGMETRICS '97*, pages 2–13, June 1997.

[11] A. Banerjea, D. Ferrari, B. A. Mah, M. Moran, D. C. Verma, and H. Zhang. The Tenet real-time protocol suite: Design, implementation, and experiences. *IEEE/ACM Transaction on Networking*, 4(1):1–10, February 1996.

[12] A. Bavier, B. Montz, and L. L. Peterson. Predicting MPEG execution times. In *Proceedings of ACM SIGMETRICS '98*, pages 131–140, June 1998.

[13] A. Berger. Anpassungsverhalten von 'network-aware' Applikationen am Beispiel von Chariot. Master's thesis, ETH Zürich, September 1998. (in German).

[14] T. Berners-Lee, R. Cailliau, A. Luotonen, H. F. Nielsen, and A. Secret. The World-Wide Web. *The Communications of the ACM*, 37(8):76–82, August 1994.

[15] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext transfer protocol—HTTP/1.0. Request for Comments (RFC 1945), Internet Engineering Task Force, May 1996.

[16] V. Bharghavan and V. Gupta. A framework for application adaptation in mobile computing environments. In *Proceedings of the 21st Annual International Computer Software and Applications Conference*, pages 573–579, August 1997.

[17] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering*, 25(5):376–390, May 1998.

[18] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proceedings of IEEE Infocom '99*, pages 1300–1309, March 1999.

[19] J. Bolliger, U. Hengartner, and T. Gross. The effectiveness of end-to-end congestion control mechanisms. Technical Report 313, Dept. Computer Science, ETH Zürich, February 1999.

[20] J-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *Proceedings of ACM SIGCOMM '94*, pages 58–67, August 1994.

[21] G. E. P. Box, G. M. Jenkins, and G. Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, Inc., 3rd edition, 1994.

[22] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM '94*, pages 24–35, August 1994.

[23] T. Bray. Measuring the Web. In *Proceedings of 5th International World Wide Web Conference*, May 1996. also appeared in *Computer Networks and ISDN Systems*, Vol 28, 1996, 993–1005.

[24] L. Breslau and S. Shenker. Best-effort versus reservations: A simple comparative analysis. In *Proceedings of ACM SIGCOMM '98*, pages 3–16, August 1998.

[25] CAIDA measurement tool taxonomy. Cooperative Association of Internet Data Analysis (CAIDA). `http://www.caida.org/Tools/taxonomy.html`.

[26] A. Campbell, G. Coulson, F. Garcia, and D. Hutchison. A continuous media transport and orchestration service. In *Proceedings of ACM SIGCOMM '92*, pages 99–110, August 1992.

[27] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proceedings of IEEE Infocom 2000*, pages 1742–1751, March 2000.

[28] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Computer Science Department, Boston University, March 1996.

[29] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. Technical Report BU-CS-96-006, Computer Science Department, Boston University, March 1996.

[30] S. Cen. *A Software Feedback Toolkit and its Application in Adaptive Multimedia Systems*. PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, October 1997.

[31] S. Cen, C. Pu, R. Stähli, C. Cowan, and J. Walpole. A distributed real-time MPEG video audio player. In *Proceedings of 5th International Workshop on Network and Operating Systems Support for Digital Video and Audio (NOSSDAV '95)*, pages 151–162, April 1995.

[32] Common gateway interface (CGI). `http://www.w3.org/CGI`.

[33] S. Chandra and C. Ellis. JPEG compression metric as a quality-aware image transcoding. In *Proceedings of 2nd Symposium on Internet Technologies and Systems (USITS '99)*, October 1999.

[34] S. Chandra, C. Ellis, and A. Vahdat. Multimedia Web services for mobile clients using quality aware transcoding. In *Proceedings of the 2nd ACM International Workshop on Wireless mobile multimedia (WoWMoM '99)*, pages 99–108, August 1999.

[35] D. D. Clark. The structuring of systems using upcalls. In *Proceedings of the 10th ACM Symposium of Operating System Principles*, pages 171–180, December 1985.

[36] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of ACM SIGCOMM '88*, pages 106–114, August 1988.

[37] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *Proceedings of ACM SIGCOMM '92*, pages 14–26, August 1992.

[38] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of ACM SIGCOMM '90*, pages 200–208, September 1990.

[39] R. De Silva, B. Landfeldt, S. Ardon, A. Seneviratne, and C. Diot. Managing application level quality of service through TOMTEN. *Computer Networks*, 31(7):727–739, April 1999.

[40] A. Dimai. *Invariant Scene Representation for Preattentive Similarity Assessment*. PhD thesis, ETH Zürich, May 1999. No. 13164.

[41] P. Dinda. The statistical properties of host load. *Scientific Programming*, 7(3–4):211–229, Winter 1999.

[42] P. Dinda and D. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings 8th IEEE Symposium on High-Performance Distributed Computing (HPDC-8)*, August 1999.

[43] P. Dinda and D. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.

[44] A. B. Downey. Using pathchar to estimate Internet link characteristics. In *Proceedings of ACM SIGCOMM '99*, pages 241–250, August 1999.

[45] G. Fankhauser, M. Dasen, N. Weiler, B. Plattner, and B. Stiller. Wavevideo—an integrated approach to adaptive wireless video. *ACM Monet, Special Issue on Adaptive Mobile Networking and Computing*, 4(4):255–271, December 1999.

[46] FastCGI. `http://www.fastcgi.com`.

[47] Z. Fei, S. Bhattacharjee, E. W. Zegura, and M. H. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proceedings of IEEE Infocom '98*, March 1998.

[48] K. W. Fendick. Evolution of controls for the available bit rate service. *IEEE Communications Magazine*, 34(11):35–39, November 1996.

[49] D. Ferrari, A. Banerjea, and H. Zhang. Network support for multimedia: A discussion of the Tenet approach. *Computer Networks and ISDN Systems*, 10:1267–1280, July 1994.

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. Request for Comments (RFC 2068), Internet Engineering Task Force, January 1997.

[51] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, B. Dom, Q. Huang, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the QBIC system. *IEEE Computer*, 28(9):23–32, 1995.

[52] S. Floyd. Connections with multiple congested gateways in packet-switched networks, part 1: One-way traffic. *Computer Communications Review*, 21(5):30–47, October 1991.

[53] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999.

[54] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4), August 1993.

[55] A. M. Ford. *Relations between Image Quality and Still Image Compression*. PhD thesis, University of Westminster, UK, May 1997.

[56] M. Fowler and K. Scott. *UML Distilled: Applying the standard object modeling language*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1997.

[57] A. Fox. *A Framework for Separating Server Scalability and Availability from Internet Application Functionality*. PhD thesis, University of California, Berkeley, CA, 1998.

[58] A. Fox and E. A. Brewer. Reducing WWW latency and bandwidth requirements via real-time distillation. In *Proceedings of the 5th International World Wide Web Conference*, May 1996.

[59] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *ASPLOS-VII Proceedings, 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–173, October 1996.

[60] A. Fox, S. D. Gribble, Y. Chawathe, and E. A. Brewer. TranSend Web accelerator proxy, 1997. `http://transend.cs.berkeley.edu`.

[61] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th ACM Symposium of Operating System Principles*, pages 78–91, October 1997.

[62] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniewicz, and Y. Jin. An architecture for a global Internet host distance estimation service. In *Proceedings of IEEE Infocom '99*, pages 210–217, March 1999.

[63] H. Frystyk Nielsen, J. Gettys, A. Barid-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of ACM SIGCOMM '97*, pages 155–166, September 1997.

[64] E. Fulp, M. Ott, D. Reininger, and D. S. Reeves. Paying for QoS: An optimal distributed algorithm for pricing network resources. In *Proceedings of 6th International Workshop on Quality of Service (IWQoS '98)*, pages 75–84, May 1998.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, December 1994.

[66] J. M. Gilbert and R. W. Brodersen. Globally progressive interactive Web delivery. In *Proceedings of IEEE Infocom '99*, pages 1291–1299, March 1999.

[67] T. Gross, P. Steenkiste, and J. Subhlok. Adaptive distributed applications on heterogeneous networks. In *8th Heterogeneous Computing Workshop*, pages 209–218, April 1999.

[68] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. Request for Comments (RFC 2608), Internet Engineering Task Force, June 1999.

[69] R. Han. Factoring a mobile client's effective processing speed into the image transcoding decision. In *Proceedings of the 2nd ACM International Workshop on Wireless mobile multimedia (WoWMoM '99)*, pages 91–98, August 1999.

[70] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. Dynamic adaptation in an image transcoding proxy for mobile Web browsing. *IEEE Personal Communications*, 5(6):8–17, December 1998.

[71] J. Heidemann, K. Obraczka, and J. Touch. Modeling the performance of HTTP over several transport protocols. *IEEE/ACM Transactions on Networking*, 5(5):616–645, October 1997.

[72] M. Hemy, U. Hengartner, P. Steenkiste, and T. Gross. MPEG system streams in best-effort networks. In *Proceedings of PacketVideo '99*, April 1999.

[73] H. H. Houh, J. F. Adam, M. Ismert, C. J. Lindblad, and D. L. Tennenhouse. The vunet desk area network: Architecture, implementation, and experience. *IEEE Journal of Selected Areas in Communications*, 13(4):710–721, May 1995.

[74] HyperText Markup Language. `http://www.w3.org/MarkUp`.

[75] P. Huang. *Enabling Large-Scale network Simulation: A Selective Abstraction Approach*. PhD thesis, Computer Science Department, University of Southern California, September 1999.

[76] H. Hüni, R. Johnson, and R. Engel. A framework for network protocol software. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95)*, pages 358–369, October 1995.

[77] IJG's JPEG software release 6a. Independent JPEG Group, February 1996.

[78] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1992.

[79] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–329, August 1988.

[80] V. Jacobson. Pathchar: A tool to infer characteristics of Internet paths, April 1997. Available from `ftp://ftp.ee.lbl.gov/pathchar`.

[81] V. Jacobson and S. McCanne. vat—audio teleconferencing tool, 1992. `ftp://ftp.ee.lbl.gov/conferencing/vat`.

[82] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[83] S. Jamin, P. B. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. *IEEE/ACM Transactions on Networking*, 5(1):56–70, February 1997.

[84] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. On the placement of Internet instrumentation. In *Proceedings of IEEE Infocom 2000*, pages 295–304, March 2000.

[85] Java Servlet API 2.1, 1999. `http://java.sun.com/products/servlet`.

[86] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[87] Junkbusters. `http://www.junkbusters.com`.

[88] P. Karn and C. Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM Transactions on Computer Systems*, 9(4):364–373, November 1991.

[89] R. H. Katz and E. A. Brewer. The case for wireless overlay networks. In *SPIE Multimedia and Networking Conference*, January 1996.

[90] R. Keller, S. Choi, D. Decasper, M. Dasen, G. Fankhauser, and B. Plattner. An active router architecture for multicast video distribution. In *Proceedings of IEEE Infocom 2000*, pages 1137–1146, March 2000.

[91] C. Kent and J. Mogul. Fragmentation considered harmful. In *Proceedings of ACM SIGCOMM '87*, pages 390–401, August 1987.

[92] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM '91*, pages 3–15, September 1991.

[93] B. Knutsson and M. Björkman. Adaptive end-to-end compression for variable-bandwidth communication. *Computer Networks*, 31(7):767–779, April 1999.

[94] K. Lai and M. Baker. Measuring bandwidth. In *Proceedings of IEEE Infocom '99*, pages 235–245, March 1999.

[95] T. V. Lakshman and U. Madhow. The performance of TCP/IP for networks with high bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking*, 5(3):336–350, June 1997.

[96] W. E. Leland, W. Willinger, M. S. Taqqu, and D. V. Wilson. On the self-similar nature of Ethernet traffic. In *Proceedings of ACM SIGCOMM '93*, pages 183–193, September 1993.

[97] D. Lin and H. T. Kung. TCP fast recovery strategies: Analysis and improvements. In *Proceedings of IEEE Infocom '98*, March 1998.

[98] L. Lippert and M. Gross. Fast wavelet based volume rendering by accumulation of transparent texture maps. In *Proceedings of the Eurographics '95*, pages 431–444, August 1995.

[99] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proceedings of 7th IEEE Symposium on High-Performance Distributed Computing (HPDC-7)*, pages 189–196, July 1998.

[100] B. Lowekamp, D. O'Hallaron, and T. Gross. Direct network queries for discovering network resource properties in a distributed environment. In *Proceedings of 8th IEEE Symposium on High-Performance Distributed Computing (HPDC-8)*, August 1999.

[101] M. Mathis. TReno bulk transfer capacity. Internet Draft, February 1999. `http://www.ietf.org/internet-drafts/draft-ietf-ippm-treno-btc-03.txt`.

[102] M. Mathis and M. Allman. Empirical bulk transfer capacity. Internet Draft, October 1999. `http://www.ietf.org/internet-drafts/draft-ietf-ippm-btc-framework-02.txt`.

[103] M. Mathis and S. Floyd. TCP-friendly unicast rate-based flow control. Technical note sent to the end2end-interest mailing list, January 1997. `http://www.psc.edu/networking/papers/tcp_friendly.html`.

[104] M. Mathis and J. Mahdavi. Diagnosing Internet congestion with a transport layer performance tool. In *Proceedings of INET 96*, June 1996.

[105] M. Mathis and J. Mahdavi. Forward acknowledgment: Refining TCP congestion control. In *Proceedings of ACM SIGCOMM '96*, pages 281–292, August 1996.

[106] M. Mathis and J. Mahdavi. TCP rate-halving with bounding parameters. `http://www.psc.edu/networking/papers/FACKnotes/current`, October 1996.

[107] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP selective acknowledgment options. Request for Comments (RFC 2018), Internet Engineering Task Force, October 1996.

[108] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review*, 27(3):67–82, July 1997.

[109] S. McCanne. *Scalable Compression and Transmission of Internet Multicast Video*. PhD thesis, University of California Berkeley, December 1996.

[110] S. McCanne, E. Brewer, R. H. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T.-L. Tung, D. Wu, and B. Smith. Toward a common infrastructure for multimedia-networking middleware. In *Proceedings 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[111] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, January 1993.

[112] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *Proceedings of ACM Multimedia '95*, pages 511–522, November 1995.

[113] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM '96*, pages 117–130, August 1996.

[114] S. McCanne and C. Torek. A randomized sampling clock for CPU utilization estimation and code profiling. In *Proceedings of the 1993 Winter USENIX Technical Conference*, January 1993.

[115] S. McCanne, M. Vetterli, and V. Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE Journal on Selected Areas in Communications*, 16(6):983–1001, August 1997.

[116] M. McIlhagga, A. Light, and I. Wakeman. Towards a design methodology for adaptive applications. In *Proceedings of ACM Mobicom '98*, pages 133–144, October 1998.

[117] J. K. McKie-Maxon and H. R. Varian. Pricing congestible network resources. *IEEE Journal of Selected Areas in Communications*, 13(7):1141–1149, September 1995.

[118] J. K. McKie-Maxon and H. R. Varian. Pricing the Internet. In B. Kahin and J. Keller, editors, *Public Access to the Internet*, pages 269–314. MIT Press, 1995.

[119] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quaterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1996.

[120] G. B. Mehta. Preference and utility. In S. Barbera, P. J. Hammond, and C. Seidl, editors, *Handbook of Utility Theory: Principles*, pages 1–47. Kluwer Academic Publishers, 1998.

[121] N. Miller and P. Steenkiste. Collecting network status information for network-aware applications. In *Proceedings of IEEE Infocom 2000*, pages 641–650, March 2000.

[122] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277, 1968.

[123] J. L. Mitchell, W. B. Pennebaker, C. E. Fogg, and D. J. LeGall. MPEG video compression standard. In *Digital Multimedia Standards Series*. Chapman and Hall, 1997.

[124] R. Mohan, J. R. Smith, and C.-S. Li. Adapting multimedia Internet content for universal access. *IEEE Transactions on Multimedia*, 1(1):104–114, March 1999.

[125] R. Morris. TCP behavior with many flows. In *Proceedings of ICNP 97*, October 1997.

[126] A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the Internet. In *Proceedings of IEEE Infocom '99*, pages 304–312, March 1999.

[127] M. Näf. Ein adaptives Bildtransferprotokoll für Chariot. Internal report, Dept. Computer Science, ETH Zürich, March 1997. (in German).

[128] K. Nahrstedt and J. M. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[129] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE Computer*, 28(5):52–63, May 1995.

[130] B. D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1998.

[131] B. D. Noble, L. Li, and A. Prakash. The case for better throughput estimation. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HoTOS '99)*, page MISSING, March 1999.

[132] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium of Operating System Principles*, pages 276–287, October 1997.

[133] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM '97*, pages 51–62, September 1997.

[134] T. Ott, J. Kemperman, and M. Mathis. The stationary behavior of ideal TCP congestion avoidance, August 1996. `ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps`.

[135] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1994.

[136] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proceedings of ACM SIGCOMM '98*, pages 303–314, August 1998.

[137] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *ASPLOS-VIII Proceedings, 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, October 1998.

[138] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

[139] C. Partridge. *Gigabit Networking*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, December 1993.

[140] V. Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of ACM SIGCOMM 97*, pages 167–179, September 1997.

[141] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California and Lawrence Berkeley National Laboratory, Berkeley, April 1997.

[142] V. Paxson. On calibrating measurements of packet transit times. In *Proceedings of ACM SIGMETRICS '98*, pages 11–21, June 1998.

[143] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[144] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. Request for Comments (RFC 2330), Internet Engineering Task Force, May 1998.

[145] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale Internet measurement. *IEEE Communications*, 36(8):48–54, August 1998.

[146] J. Poskanzer. PBMPLUS: Extended portable bitmap toolkit, December 1993. NetPBM Release 7.

[147] Proxynet. `http://www.proxynet.com`.

[148] Intel: QuickWeb. `http://www.intel.com./quickweb`, (discontinued).

[149] K. K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. In *Proceedings of ACM SIGCOMM '88*, pages 303–313, August 1988.

[150] S. Raman and S. McCanne. A model, analysis and protocol framework for soft state-based communication. In *Proceedings of ACM SIGCOMM '99*, pages 15–25, August 1999.

[151] Real Networks. `http://www.real.com`.

[152] R. Rejaie, D. Estrin, and M. Handley. Quality adaptation for congestion controlled video playback over the Internet. In *Proceedings of ACM SIGCOMM '99*, pages 189–200, August 1999.

[153] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *Proceedings of IEEE Infocom '99*, pages 1337–1345, March 1999.

[154] O. Rubin. *The Design of Automatic Control Systems.* Artech House, Norwood, MA, 1986.

[155] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[156] A. Sang and S. Li. A predictability analysis of network traffic. In *Proceedings of IEEE Infocom 2000*, pages 342–351, March 2000.

[157] M. Satyanarayanan. RPC2 user guide and reference manual, October 1991. `http://www.coda.cs.cmu.edu/doc/html/rpc2_manual.html`.

[158] M. Satyanarayanan. Fundamental challenges in mobile computing. In *15th Symposium on Principles of Distributed Computing*, pages 1–7, May 1996.

[159] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *Proceedings of ACM SIGCOMM '99*, pages 289–299, August 1999.

[160] P. Schneck and K. Schwan. Dynamic authentication for high-performance networked applications. In *Proceedings of 6th International Workshop on Quality of Service (IWQoS '98)*, pages 127–136, May 1998.

[161] H. Schulzrinne. Operating system issues for continuous media. *Multimedia Systems*, 4:269–280, 1996.

[162] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. Request for Comments (RFC 1889), Internet Engineering Task Force, January 1996.

[163] S. Seshan, M. Stemm, and R. H. Katz. SPAND: Shared passive network performance discovery. In *USENIX 97 Symposium on Internet Technologies and Systems*, December 1997.

[164] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technology Journal*, 27:379–423, 1948.

[165] S. Shenker. Making greed work in networks: A game-theoretic analysis of switch service disciplines. In *Proceedings of ACM SIGCOMM '94*, pages 47–57, August 1994.

[166] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service. Request for Comments (RFC 2212), Internet Engineering Task Force, September 1997.

[167] S. Shenker and J. Wroclawski. Network element service specification template. Request for Comments (RFC 2216), Internet Engineering Task Force, September 1997.

[168] B. Siegell and P. Steenkiste. Automatic selection of load balancing parameters using compile-time and run-time information. *Concurrency—Practice and Experience*, 9(3):275–317, 1996.

[169] B. C. Smith. *Implementation Techniques for Continuous Media Systems and Applications*. PhD thesis, University of California, Berkeley, December 1994.

[170] W. Stallings. *SNMP, SNMPv2, and CMIP: The practical guide to network management*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1993.

[171] P. Steenkiste. Adaptation models for network-aware distributed computations. In A. Sivasubramaniam and M. Lauria, editors, *Network-based Parallel Computing Communication, Architecture, and Applications*, volume 1602, pages 16–31. Springer-Verlag, January 1999.

[172] R. Steinmetz. Analyzing the multimedia operating system. *IEEE Multimedia*, 2(1):68–84, Spring 1995.

[173] M. Stemm. *A Network Measurement Architecture for Adaptive Applications*. PhD thesis, University of California, Berkeley, CA, 1999.

[174] M. Stemm, R. Katz, and S. Seshan. A network measurement architecture for adaptive applications. In *Proceedings of IEEE Infocom 2000*, pages 285–294, March 2000.

[175] A. Stepanov and M. Lee. The standard template library. Technical report, Hewlett-Packard Laboratories, June 1995.

[176] R. W. Stevens. *TCP/IP Illustrated: The Protocols*, volume 1. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, December 1993.

[177] B. Stiller, G. Fankhauser, B. Plattner, and N. Weiler. Charging and accounting for integrated Internet services—state of the art, problems, and trends. In *Proceedings of INET 98*, pages 21–24, June 1998.

[178] M. Stricker and A. Dimai. Color indexing with weak spatial constraints. In *Storage and Retrieval for Image and Video Databases IV*, volume 2670 of *SPIE Proceedings Series*, February 1996.

[179] V. Strumpen. *The Network Machine*. PhD thesis, ETH Zürich, July 1995. No. 11227.

[180] C. A. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1997.

[181] Taligent, Inc. Building object-oriented frameworks. `http://www.taligent.com/Technology/Technology.html`, 1994.

[182] W. Theilmann and K. Rothermel. Dynamic distance maps of the Internet. In *Proceedings of IEEE Infocom 2000*, pages 275–284, March 2000.

[183] T. Turletti and C. Huitema. Videoconferencing on the Internet. *IEEE/ACM Transactions on Networking*, 4(3):340–351, June 1996.

[184] C. J. Turner and L. L. Peterson. Image transfer: An end-to-end design. In *Proceedings of ACM SIGCOMM '92*, pages 258–268, August 1992.

[185] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. Request for Comments (RFC 2165), Internet Engineering Task Force, June 1997.

[186] A. Veres and M. Boda. The chaotic nature of TCP congestion control. In *Proceedings of IEEE Infocom 2000*, pages 1715–1723, March 2000.

[187] L. Vicisano, L. Rizzo, and J. Crowcroft. TCP-like congestion control for layered multicast data transfer. In *Proceedings of IEEE Infocom '98*, pages 996–1003, March 1998.

[188] H. Wactlar, T. Kanade, M. Smith, and S. Stevens. Intelligent access to digital video: The Informedia project. *IEEE Computer*, 29(5):46–52, May 1996.

[189] P. Walther. Adaptiver Chariot Server. Master's thesis, ETH Zürich, August 1997. (in German).

[190] All Things Web. Third state of the web survey, May 1999. `http://www.pantos.org/atw/35654.html`.

[191] R. Weber, August 1999. private communication.

[192] R. Weber and J. Bolliger. Chariot messages—protocol definition, 1998. Internal document.

[193] R. Weber, J. Bolliger, T. Gross, and H.-J. Schek. Architecture of a networked image search and retrieval system. In *Proceedings 8th ACM Conference on Information and Knowledgement (CIKM '99)*, pages 430–441, November 1999.

[194] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, volume 24, pages 194–205, August 1998.

[195] M. Wechsler. *Spoken Document Retrieval Based on Phoneme Recognition*. PhD thesis, ETH Zürich, October 1998. No. 12879.

[196] D. Wetherall and C. J. Lindblad. Extending Tcl for dynamic object-oriented programming. In *Proceedings of the Tcl/Tk Workshop*, July 1995.

[197] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. In *Proceedings of ACM SIGCOMM '95*, pages 100–113, August 1995.

[198] A. Woodruff, P. M. Aoki, E. Brewer, P. Gauthier, and L. A. Rowe. An investigation of documents from the World Wide Web. In *Proceedings of 5th International World Wide Web Conference*, May 1996.

[199] G. R. Wright and R. W. Stevens. *TCP/IP Illustrated: The Implementation*, volume 2. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, January 1995.

[200] J. Wroclawski. Specification of the controlled-load network element service. Request for Comments (RFC 2211), Internet Engineering Task Force, September 1997.

[201] J. Wroclawski. The use of RSVP with IETF integrated services. Request for Comments (RFC 2210), Internet Engineering Task Force, September 1997.

[202] B. Zenel and D. Duchamp. A general purpose proxy filtering mechanism applied to the mobile environment. In *Proceedings of ACM MOBICOM '97*, pages 248–259, September 1997.

[203] L. Zhang, S. Berson, S. Herzog, and S. Jamin. ReSource reserVation Protocol (RSVP)—version 1 functional specification. Request for Comments (RFC 2205), Internet Engineering Task Force, September 1997.

[204] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(5):8–18, September 1993.

[205] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.

Curriculum Vitae

Jürg Bolliger

March 31, 1969	Born in Basel, Switzerland Son of Margrit and Walter Bolliger
1975–1981	Primary school, Baar
1981–1988	Gymnasium, Zug
1988	Matura Typus C
1988–1994	Studies in Computer Science Swiss Federal Institute of Technology, Zürich
1991	Internship at Landis & Gyr AG, Zug
1994	Diploma in Computer Science, ETH Zürich (Silver medal for diploma thesis on "Aggressive Instruction Scheduling")
1994	Visiting Researcher, Symbolic Computation Group University of Waterloo, Canada
1995–2000	Research and teaching assistant Institute of Computer Systems, ETH Zürich in the research group of Prof. Dr. T. Gross