THÈSE DE DOCTORAT

de

L'UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS

Spécialité

SYSTÈMES INFORMATIQUES

présentée par

Jussi Antti Tapio KANGASHARJU

pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ DE NICE – SOPHIA ANTIPOLIS

Sujet de la thèse :

Distribution de l'information sur Internet
(Internet Content Distribution)

Soutenu le 26 avril 2002 à 14h30
à Institut Eurécom – Sophia Antipolis
devant le jury composé de :

Président	Dr. Michel Riveill	Professeur, Université de Nice Sophia Antipolis
Rapporteurs	Dr. Andrzej Duda	Professeur, Institut d'Informatique et de Mathématiques Appliquées de Grenoble, Grenoble
	Dr. Mesaac Makpangou	Chargé de recherche, INRIA, Rocquencourt
Examinateurs	Dr. Philippe Nain	Directeur de recherche, INRIA, Sophia Antipolis
	Dr. James Roberts	France Télécom R&D, Issy les Moulineaux
Directeur de Thèse	Dr. Keith W. Ross	Professeur, Institut Eurécom, Sophia Antipolis

Résumé

Cette thèse étudie la distribution de contenu sur Internet. Dans la première partie nous étudions les méthodes de redirection des clients. Nous développons une architecture pour localiser les copies d'un objet. Cette architecture est une extension du Domain Name System et peut être mise en place d'une manière incrémentale. Nous présentons une architecture d'annuaire répliqué et montrons comment réaliser le Domain Name System avec cette architecture. Cette architecture permet de stocker des informations qui changent rapidement, elle peut être réalisée de manière incrémentale, et ne nécessite aucun changement logiciel. L'évaluation de performance de cette architecture nous donne des indications sur la durée pendant laquelle on peut cacher l'information. Nous évaluons aussi les performance des méthodes de redirection utilisées par les réseaux de distribution de contenu modernes. Nos résultats montrent que le coût associé à l'ouverture de nouvelles connexions peut limiter sévèrement les performances perçues par l'utilisateur.

Dans la deuxième partie nous considérons la réplication d'objets. Nous développons un modèle d'optimisation combinatoire pour répliquer des objets dans un réseau de distribution. Nos résultats montrent que la meilleure performance est obtenue quand la réplication est coordonnée sur tout le réseau. Nous étudions la réplication optimale de contenu dans les réseaux de type peer-to-peer. Nous construisons un modèle et développons plusieurs algorithmes adaptatifs pour répliquer les objets de manière dynamique. Nos résultats montrent que nos algorithmes, combinés avec une politique de remplacement LFU, offrent une performance presque optimale. Nous considérons aussi la distribution de vidéos en couches en utilisant un modèle de "knapsack" stochastique. Nous développons plusieurs heuristiques pour déterminer quelles couches de quelles vidéos doivent être cachées afin de maximiser le revenu.

Abstract

This thesis studies Internet content distribution. In the first part of the thesis we consider client redirection mechanisms. We develop an architecture for locating copies of cached objects. This architecture is a small extension to the Domain Name System and can be deployed incrementally. We present an architecture of an Internet-wide replicated directory service and show how the current Domain Name System can be implemented with this architecture. The key features of this architecture are that it allows us to store rapidly changing information, can be deployed incrementally, and requires no changes to existing software. Our extensive performance evaluation of this architecture provides us with insight on how long the information can be cached. We also evaluate the performance of the redirection mechanisms used by modern content distribution networks. We find that the overhead of opening new connections to new servers can severly limit the user-perceived performance.

In the second part of the thesis we consider object replication in content distribution. We develop a combinatorial optimization model for optimally replicating objects in a content distribution network. Our results show that best performance is obtained when replication is coordinated over the whole network. Using the same model we also develop cooperation strategies for peer-to-peer networks. We also consider the problem of optimal content replication in peer-to-peer communities. We formulate this problem as an integer programming problem and develop several adaptive algorithms to replicate objects on-the-fly. Our results indicate that our algorithms combined with least-frequently-used replacement policy provide near-optimal performance. We also consider the distribution of layered encoded video using a stochastic knapsack model. We develop several heuristics to determine which layers of which videos should be cached in order to maximize the accrued revenue.

Acknowledgements

First and foremost, I would like to thank Prof. Keith Ross, my advisor, for all the support and encouragement he has given me, and also for giving me the possibility to do my thesis here at Eurécom.

I would also like to thank Reza Rejaie, from AT&T Labs – Research, for the two excellent internships I did at AT&T in Menlo Park.

Big thanks to Albert, Alex, Chris, Cristina, Daniela, David, Despina, Max, and Patrick, my friends and office-mates at Eurécom, for all the fun we've had during the past three years, both in and out of the office. Special thanks also to the four brave fish we had in our office: Bob, Quincy, Buzz, and Woody. I know you guys have been through a lot because of us and I appreciate your sacrifices.

Thanks also to Carine and Philippe for reading through and correcting the French parts of my thesis.

I am also grateful to French Ministry of Education, France Télécom, and the Finnish Cultural Foundation who provided the funding necessary for allowing me to work in the best of conditions.

The logfiles used in Section 3.3.1 were provided by National Science Foundation (grants NCR-9616602 and NCR-9521745), and the National Laboratory for Applied Network Research. The routing data summaries used in constructing the network topologies in Chapter 6 were provided by National Science Foundation Cooperative Agreement No. ANI-9807479, and the National Laboratory for Applied Network Research.

Finally, I would like to express my profound gratitude towards my family who have always supported me during my studies.

Contents

Introduction (en français) 19

1 Introduction 25
1.1 Thesis Contributions . . . 26
1.2 Thesis Organization . . . 28
1.3 Published Work . . . 28

2 Overview of Content Distribution 31
2.1 World Wide Web . . . 31
2.2 Proxy Caching . . . 32
2.2.1 Caching Hierarchies . . . 34
2.2.2 Problems with Client-side Caching . . . 36
2.3 Content Distribution Networks . . . 36
2.3.1 Full Replication . . . 37
2.3.2 Partial Replication . . . 38
2.4 Peer-to-Peer Networks . . . 38
2.4.1 Centralized Architecture . . . 39
2.4.2 Distributed Architecture . . . 40
2.4.3 Hybrid Architectures . . . 42
2.5 Conclusion . . . 42

I Client Redirection 43

3 Locating Copies of Objects 45
3.1 Overview . . . 45
3.2 Introduction . . . 45
3.3 Location Data System (LDS) . . . 47
3.3.1 Resource Records, Query Messages and Reply Messages . . . 49
3.3.2 Updating the Location Servers . . . 51
3.3.3 Implementation . . . 51
3.4 Network Web Caching . . . 52
3.4.1 Populating Caches . . . 53
3.4.2 Network Traffic . . . 53

3.4.3 Practical Considerations . . . 54
3.5 Replicated Servers . . . 55
3.5.1 Reducing LDS Query Traffic . . . 55
3.6 Routing Decision . . . 56
3.7 Related Research . . . 57
3.8 Future Work . . . 58
3.9 Conclusion . . . 58

4 Replicated Directory Service 61
4.1 Overview . . . 61
4.2 Introduction . . . 61
4.3 Overview of DNS . . . 62
4.4 Replicated DNS Architecture . . . 64
4.4.1 Storing the Entire Database on a Server . . . 64
4.4.2 Interaction between Authoritative and Replicated Nameservers . . . 65
4.4.3 Interaction among Replicated Nameservers . . . 66
4.4.4 Traffic Analysis for Communication among Replicated Servers . . . 68
4.4.5 Resolving DNS Queries . . . 68
4.4.6 Arpanet Name Resolution . . . 69
4.5 Staleness vs. Traffic at Replicated Nameservers . . . 70
4.6 Latency Improvement . . . 76
4.6.1 Unpopular Servers . . . 77
4.6.2 Popular Servers . . . 77
4.6.3 Replicated Server . . . 77
4.7 Graceful Migration from Distributed to Replicated System . . . 80
4.7.1 Gathering Information . . . 81
4.8 Security Issues and Fault Tolerance . . . 82
4.8.1 Fault Tolerance . . . 83
4.9 Conclusion . . . 84

5 Client Redirection Performance 85
5.1 Overview . . . 85
5.2 Introduction . . . 85
5.2.1 Related Work . . . 86
5.3 Redirection to Servers . . . 87
5.4 Simulation Model . . . 87
5.4.1 Redirection Schemes . . . 88
5.4.2 Files . . . 90
5.5 Simulation Results . . . 90
5.5.1 Loss Free Conditions . . . 91
5.5.2 Simulations with Loss in Network . . . 92
5.6 Experiments . . . 95
5.7 Discussion . . . 97

5.8 Conclusion 98

II Object Replication 101

6 Replication in Content Distribution Networks 103
6.1 Overview 103
6.2 Introduction 103
6.3 Network Model 104
6.3.1 Network Topologies 104
6.4 Cost Model 105
6.4.1 Proving NP-Completeness 106
6.5 Replication Heuristics 107
6.6 Evaluation of Heuristics 108
6.7 Peer-to-Peer Content Distribution 112
6.8 Related Work 114
6.9 Conclusion 116

7 Replication in Peer-to-Peer Communities 119
7.1 Overview 119
7.2 Introduction 119
7.3 Related Work 122
7.4 Optimal Replication of Content over a Community of Peers 123
7.5 Adaptive Replication for Content Caches 125
7.5.1 Distributed LFU Replacement 130
7.5.2 Timescales 136
7.6 Adaptive Replication for Content Clubs 136
7.7 Dealing with Hot Spots 140
7.8 Conclusion 143

8 Distribution of Layered Encoded Video 145
8.1 Overview 145
8.2 Introduction 145
8.3 Model of Layered Video Streaming with Proxy 146
8.3.1 Layered Video 147
8.3.2 Proxy Server 148
8.3.3 Stream Delivery 149
8.4 Optimal Caching 150
8.4.1 Utility Heuristics 151
8.4.2 Evaluation of Heuristics 152
8.5 Negotiation about Stream Quality 158
8.5.1 Numerical Results 159
8.6 Queueing of Requests 159
8.7 Is Partial Caching Useful ? 161

8.7.1 Numerical Results . . . 162
8.8 Related Work . . . 164
8.9 Conclusion . . . 164

9 Conclusion **167**
9.1 Summary . . . 167
9.2 Directions for Future Work . . . 169

Conclusion (en français) **171**

Bibliography **175**

A Résumé de la thèse en français **183**
A.1 Introduction . . . 183
A.2 Distribution de l'information . . . 183
A.3 Localisation des objets . . . 184
A.3.1 Motivation . . . 184
A.3.2 Location Data System (LDS) . . . 184
A.4 Réplication de DNS . . . 185
A.4.1 Motivation . . . 185
A.4.2 Architecture . . . 185
A.5 Redirection dans les CDNs . . . 186
A.5.1 Motivation . . . 186
A.5.2 Résultats . . . 187
A.6 Réplication dans les réseaux CDN . . . 188
A.6.1 Motivation . . . 188
A.6.2 Résultats . . . 188
A.7 Les communautés peer-to-peer . . . 189
A.7.1 Motivation . . . 189
A.7.2 Résultats . . . 190
A.8 Vidéo en couches et les caches . . . 191
A.8.1 Motivation . . . 192
A.8.2 Approche . . . 193

List of Tables

4.1 Number of updates per host per week for different satellite links 68
4.2 DNS query results for unpopular servers 77
4.3 DNS query results for popular servers 80

5.1 Different pages used in simulations 90
5.2 Results from experiments 97

6.1 AS topologies 105

8.1 Utility definitions. 152
8.2 Average error of heuristics in small problems 153

List of Figures

2.1 Client, Proxy, and Server 33
2.2 Caching hierarchy 35
2.3 Content distribution network 37
2.4 Centralized peer-to-peer network 40
2.5 Distributed peer-to-peer network 41

3.1 LDS service 48
3.2 Sequence of Location Queries and Replies 49
3.3 Representation of a URL 49

4.1 A DNS query 63
4.2 Evolution of Hard Disk Storage Capacity 64
4.3 Connections between authoritative nameservers (AS) and replicated nameservers (RNS) 66
4.4 Clients, Local Nameserver (NS), and Replicated NS 71
4.5 Client Requests and Modification Times 71
4.6 Fraction of queries forwarded to replicated nameserver 73
4.7 Fraction of stale resource records delivered to clients 74
4.8 Fraction of stale resource records delivered to clients (continued) 75
4.9 Comparing fraction stale RRs and fraction queries to replicated server . . . 76
4.10 DNS query latency histograms for unpopular servers 78
4.11 DNS query latency histograms for popular servers 79

5.1 Simulation model 88
5.2 Effect of bandwidth on download time 89
5.3 Performance of the full redirection scheme 91
5.4 Selective redirection with parallel connections 93
5.5 Selective redirection with pipelining 94
5.6 Performance of the full redirection scheme with loss 95
5.7 Selective redirection with parallel connections with loss 96

6.1 Experiments with 1,000 objects 110
6.2 Experiments with 10,000 objects 111
6.3 Gain from cooperation for $K = 50$, 1000 objects 115
6.4 3-way cooperation 116

6.5 Gain from cooperation for $D_{AB} = 2$, $K = 50$, 1000 objects 117

7.1 Benefits of coordinated replication 122
7.2 Hit-rate as function of node storage capacity for Zipf parameter .8 128
7.3 Hit-rate as function of node storage capacity for Zipf parameter 1.2 129
7.4 Fraction of misses for Zipf = .8 130
7.5 Number of replicas per object with 10 objects of per-node storage capacity and LRU replacement policy for Zipf parameter .8 131
7.6 Number of replicas per object with 10 objects of per-node storage capacity and LRU replacement policy for Zipf parameter 1.2 132
7.7 Number of replicas per object with 10 objects of per-node storage capacity and LFU replacement policy for Zipf parameter .8 134
7.8 Number of replicas per object with 10 objects of per-node storage capacity and LFU replacement policy for Zipf parameter 1.2 135
7.9 Sensitivity of hit probabilities to request rate 136
7.10 Hit-rate as function of node storage capacity for Zipf = .8 and $L = 5$ 138
7.11 Hit-rate as function of node storage capacity for Zipf = .8 and $L = 10$ . . . 139
7.12 Hit-rate as function of L and K for Zipf = .8 and 30 objects of per-node storage 140
7.13 Hit-rate as function of L and K for Zipf = .8 and 160 objects of per-node storage 141

8.1 Architecture for caching and streaming of layered encoded video. 147
8.2 Revenue as function of link capacity for 3 different cache sizes 154
8.3 Revenue as function of cache size for 2 different link capacities 155
8.4 Revenue as function of cache size and link capacity 156
8.5 $1 - B(\mathbf{c})$ as function of cache size and link capacity 156
8.6 Effect of Zipf-parameter ζ on revenue 157
8.7 Effect of client request rate on revenue 157
8.8 Increased revenue from renegotiation 160
8.9 Increased revenue from queueing requests for buffer size of 100 160
8.10 Normalized throughput for partial caching and trunk reservation with $C =$ 150 Mbps 163
8.11 Normalized throughput for partial caching and trunk reservation with different Zipf parameters 163

A.1 Codage d'URL en DNS 184
A.2 Comparaison de fraction de réponses incorrectes à la fraction de requêtes . . 186
A.3 Performance de deux méthodes de redirection 187
A.4 Performance des heuristiques de placement 189
A.5 Avantages de coopération 190
A.6 Taux de hit 191
A.7 Nombre de copies 192
A.8 Taux de hit dans un club de contenu 192

A.9 Performance des heuristiques 193

Introduction

Ces dernières années l'Internet et le World Wide Web sont devenus très populaires. L'Internet est devenu une source importante d'information et de divertissement pour des personnes partout dans le monde. Cette croissance phénoménale a été incitée par la facilité d'utilisation des "browsers", et l'ajout d'un contenu plus attrayant, tel que l'audio et la vidéo. Plus les gens auront un accès rapide chez eux, comme l'ADSL et le câble, plus la demande de contenu de haute qualité augmentera. En outre, comme les réseaux sans fil, tels que les téléphones portables et les PDAs, deviennent plus populaires, nous pourrons accéder à l'Internet de n'importe où et à n'importe quel moment.

Cette demande toujours croissante place une charge élevée sur l'infrastructure de l'Internet. Les serveurs et les liaisons du réseau doivent supporter une demande fortement variable et imprévisible. Certains contenus, tels que les portails principaux comme Yahoo [94], MSN [49], ou Netscape [54], sont constamment populaires, et l'opérateur du site peut prendre des mesures pour augmenter sa disponibilité. La répartition de la charge sur plusieurs serveurs et l'installation de liaisons de haute capacité peuvent considérablement aider à fournir un service meilleur et plus rapide.

Le problème principal est ce que l'on appelle les "hot-spots". Ceux-ci se produisent quand un certain contenu devient soudainement extrêmement populaire, en général seulement durant une courte période. Des exemples de ces "hot-spots" sont des événements sportifs, d'autres événements d'actualité, ou le lancement d'une nouvelle version d'un logiciel populaire. Tous ces événements produisent une charge très élevée sur le serveur disposant du contenu en question, mais il est difficile de prévoir exactement quelle capacité est nécessaire. Parfois ces événements sont soudains, comme par exemple quelqu'un qui publie, sur un forum de discussion, un lien sur une page Web et lorsque les membres du forum veulent accéder à cette page. Le traitement de ces "hot-spots" est le problème principal que pose la distribution de l'information sur Internet.

La solution de base, sur laquelle les architectures modernes de distribution de contenu sont fondées, est de répliquer le contenu sur plusieurs serveurs et de diriger les différents clients sur les différents serveurs. Ceci garantit que la charge sur les serveurs et les liens du réseau restent gérables et que les utilisateurs peuvent obtenir le contenu désiré en un temps raisonnable. Cela pose néanmoins deux problèmes. D'abord, il faut savoir où et comment répliquer le contenu, et ensuite, comment s'assurer que les clients peuvent trouver ce contenu répliqué.

Le but de cette thèse est d'étudier les architectures de distribution de l'information.

Cette thèse est dans deux parties. Dans la première partie nous étudions des méthodes de redirection des clients et dans la deuxième partie nous étudions des stratégies de réplication des objets.

Une redirection de client efficace devrait remplir deux conditions. D'abord, elle devrait équilibrer la charge sur l'ensemble des serveurs disponibles aussi également que possible. Ensuite, elle devrait être facile à déployer en pratique. Cette deuxième condition s'avère être bien plus stricte que la première, parce qu'elle exige que nous ne puissions pas changer le logiciel sur les clients. Il y a déjà un grand nombre de logiciels installés et il est impossible de forcer tout le monde à installer une nouvelle version du browser pour profiter des dispositifs de redirection. Ceci implique que le mécanisme de redirection devrait s'insérer de manière transparente dans la chaîne existante de requête-réponse. Une approche alternative est d'utiliser les mécanismes qui peuvent être mis en place de manière incrémentale, tels que le 'Location Data System' que nous présentons dans le chapitre 3. L'avantage des mécanismes qui s'insèrent dans la chaîne de requête-réponse, telle que la redirection DNS comme elle pratiquée par les réseaux de distribution de contenu modernes, est qu'ils sont immédiatement à la disposition de tous les clients. L'inconvénient de ces méthodes est qu'elles ne sont pas nécessairement entièrement transparentes à l'utilisateur, et elles peuvent entraîner une perte de performance.

Les mécanismes de réplication des objets sont étroitement liés aux mécanismes de redirection. Après tout, quand les objets ont été répliqués sur plusieurs serveurs, nous devons pouvoir diriger des clients vers ces serveurs. La réplication des objets est typiquement gérée par le fournisseur de contenu ou par un tiers pour le fournisseur de contenu. Par conséquent, il est possible d'utiliser des mécanismes plus élaborés qui peuvent même exiger de changer tout le logiciel sur le serveur. Ceci nous donne la possibilité de coordonner la création des copies et de les gérer en fonction de la capacité disponible sur le serveur et dans le réseau, et au fur et à mesure des requêtes des clients. En effet, comme nous montrons dans le chapitre 6, le placement coordonné des copies d'objets fournit la meilleure performance, en comparaison des méthodes qui essayent d'agir sans coordination. Avec la redirection des clients, la réplication des objets nous offre la possibilité de distribuer la charge sur plusieurs serveurs, et de garantir ainsi la livraison rapide du contenu aux utilisateurs.

Dans le chapitre 2 nous présentons un état de l'art des technologies de distribution de contenu et de leurs architectures, ainsi que leur évolution d'une architecture de base client–serveur aux réseaux de distribution de contenu actuels. Nous prêterons une attention particulière à la façon dont ces technologies ont considéré les problèmes de la re-direction et de la réplication et aux avantages et inconvénients de ces solutions.

Nous présentons à présent les contributions principales de cette thèse.

Contributions de la thèse

Cette thèse apporte plusieurs contributions. La première contribution (chapitre 3) est une architecture pour localiser des copies d'objets. Cette architecture est basée sur une extension du Domain Name System (DNS) et permet aux clients de localiser des serveurs, des miroirs, ou des caches qui contiennent une copie de l'objet demandé. Cette architecture exige seulement des extensions mineures au DNS et peut être mis en place d'une manière incrémentale. Nous proposons également des méthodes que les clients peuvent utiliser pour déterminer quelles copies sont "les meilleures" parmi toutes les copies disponibles, c'est à dire celles qui pourront être téléchargées le plus rapidement.

La deuxième contribution (chapitre 4) est une architecture nouvelle pour un service d'annuaire répliqué à l'échelle de l'Internet. Nous montrons également comment cette architecture peut être utilisée pour le Domain Name System actuel. L'atout principal de cette nouvelle architecture est qu'elle nous permet de stocker l'information qui change rapidement dans le DNS. Actuellement, en raison de l'utilisation des caches, la capacité du système DNS pour stocker de l'information qui change rapidement est très limitée. Notre architecture utilise les technologies de communication multipoint et satellite, ce qui, avec le coût faible actuel de l'espace de stockage sur disque dur, nous permet de répliquer toute la base de données DNS sur certains serveurs. Ceci nous permet de maintenir la cohérence des données qui changent rapidement sur tous les serveurs, et également de diminuer de manière significative le temps de latence des requêtes DNS. Notre architecture est complètement transparente aux clients du DNS et peut être mis en place d'une manière incrémentale sans aucun changement aux logiciels DNS déployés. Nous présentons également une analyse de performance de notre architecture.

La troisième contribution (chapitre 5) est une évaluation des mécanismes de redirection utilisés dans les réseaux de distribution de contenu modernes. Les réseaux de distribution de contenu actuels emploient le DNS pour rediriger les clients vers les serveurs de contenu. Nous pouvons distinguer deux types de redirections – la redirection totale et la redirection sélective. En utilisant des simulations et des expériences avec des serveurs sur l'Internet, nous comparons comment ces méthodes de redirection affectent la performance perçue par l'utilisateur. Nos résultats indiquent que la redirection totale apporte une performance supérieure parce qu'elle évite l'ouverture de nouvelles connexions TCP, ce qui peut considérablement dégrader la performance de la redirection sélective. Bien que ce résultat puisse sembler intuitif, notre contribution dans ce chapitre est l'évaluation quantitative et numérique de cette différence et son effet sur la performance perçue par l'utilisateur.

En tant que notre quatrième contribution (chapitre 6), nous étudions des stratégies de réplication des objets dans les réseaux de distribution de contenu. Nous formulons le problème du placement optimal des copies d'objet comme un problème d'optimisation combinatoire que nous montrons être NP-complet. Nous développons plusieurs heuristiques pour déterminer le placement des copies et, en utilisant de vraies topologies de l'Internet, nous évaluons les performances de nos heuristiques. Nos résultats indiquent que la meilleure performance est obtenue quand la réplication des objets est coordonnée à travers tout le réseau. Les stratégies qui fonctionnent seulement localement sur un nœud donnent une performance inférieure. Nous développons également un modèle de coopération pour les réseaux de type "peer-to-peer". L'évaluation de ce modèle nous montre que la coopération peut rapporter des avantages significatifs pour les pairs en coopération.

Notre cinquième contribution (chapitre 7) est un ensemble d'algorithmes pour répliquer et localiser des objets dans une communauté peer-to-peer. Nous distinguons deux types de communautés : les caches de contenu et les clubs de contenu. Dans les deux cas, nous formulons le problème comme un problème de programmation de nombre entier qui nous fournit la solution optimale. Nous fournissons plusieurs algorithmes adaptatifs et distribués pour répliquer les objets dynamiquement et nous évaluons leur performance par des simulations. Nous trouvons qu'en combinant nos algorithmes avec une politique de remplacement LFU distribuée nous pouvons obtenir une performance quasi-optimale, en termes de taux de succès (ou 'hit-rate') et en nombre de copies d'objet. Nous considérons également une version modifiée du problème initial qui ajoute des contraintes pour équilibrer la charge sur tous les nœuds.

En tant que notre sixième contribution (chapitre 8), nous étudions la distribution de vidéos encodées en couches et distribuées à travers des caches. Nous avons considéré deux ressources, la mémoire de cache et la capacité du lien du réseau, et nous avons modélisé le problème en utilisant un modèle de knapsack stochastique. Dans notre modèle, les clients peuvent demander n'importe quel nombre de couches de la vidéo, en fonction de leur connectivité au réseau. La vidéo étant encodée en couches, nous avons la contrainte que toutes les couches inférieures doivent être présentes au décodeur pour pouvoir décoder et afficher une couche donnée. Nous proposons plusieurs heuristiques pour déterminer quelles couches de quelles vidéos devraient être cachés afin de maximiser le revenu total. Nous évaluons les performances de nos heuristiques par des expériences numériques. En outre, nous considérons également deux extensions intuitives, les négociations de qualité et les files d'attente pour les requêtes, mais nos résultats indiquent qu'elles n'ont qu'un effet secondaire sur la performance globale. Nous considérons également le cas où l'on cache uniquement quelques couches basses et où tous les clients sont seulement intéressés par des vidéos de pleine qualité. Nous avons découvert qu'il n'est pas avantageux de cacher seulement quelques couches inférieures d'une vidéo lorsque les clients demandent la pleine qualité, et que nous devrions toujours cacher des vidéos complètes.

Organisation de la thèse

Cette thèse se compose d'un chapitre préliminaire qui présente une vue d'ensemble des technologies de distribution de contenu, de six chapitres de recherche en deux parties, et d'un chapitre de conclusion.

Dans la première partie de cette thèse, couvrant les chapitres 3, 4, et 5, nous nous concentrons sur la redirection des clients. Le chapitre 3 présente le Système de Localisation des Données ou "Location Data System". C'est un système conçu pour informer les clients des copies d'objets qui existent dans le réseau. Le chapitre 4 décrit notre architecture pour le Domain Name System répliqué. Nous présentons une vue globale de l'architecture, une évaluation des performances, des stratégies de migration du DNS existant, et considérons des questions de sécurité et de tolérance aux pannes. Le chapitre 5 évalue les performances des méthodes de redirection utilisées par les réseaux de distribution de contenu. Grâce à des simulations et des expériences sur l'Internet, nous évaluons les effets de la redirection sur la performance perçue par l'utilisateur.

Dans la deuxième partie de cette thèse, couvrant les chapitres 6, 7, et 8, nous nous concentrons sur la réplication des objets. Le chapitre 6 présente un modèle pour la réplication optimale des objets dans un réseau de distribution de contenu, développe plusieurs heuristiques, et évalue leur performance en utilisant de vraies topologies de l'Internet. Nous développons également un modèle de coopération pour des réseaux de type peer-to-peer. Dans le chapitre 7 nous considérons la réplication optimale de contenu dans les communautés peer-to-peer. Nous développons un modèle qui nous donne une solution optimale, et nous présentons des algorithmes adaptatifs qui peuvent être employés pour répliquer le contenu de manière dynamique. Le chapitre 8 considère le problème de cacher des vidéos encodées en couches. Nous présentons un modèle basé sur le knapsack stochastique, développons plusieurs heuristiques, et évaluons leur performance par des expériences numériques.

Chacun des six chapitres de recherche est indépendant et, en plus de la contribution principale, présente un état de l'art du sujet abordé dans le chapitre ainsi que des directions futures de recherche pour ce sujet. En conclusion, le chapitre 9 récapitule les résultats et les contributions de cette thèse.

Publications

Les chapitres 3, 4, 5, 6, 7, et 8 ont été publiés, ou sont actuellement en cours de soumission.

- J. Kangasharju, K. W. Ross, et J. W. Roberts. Locating copies of objects using the domain name system. *Proceedings of 4th Web Caching Workshop*, San Diego, CA, mars 31 – avril 2, 1999.
- J. Kangasharju et K. W. Ross. A replicated architecture for the domain name system. *Proceedings of IEEE Infocom*, Tel Aviv, Israël, mars 26–30 2000.
- J. Kangasharju, K. W. Ross, et J. W. Roberts. Performance evaluation of redirection schemes in content distribution networks. *Proceedings of 5th Web Caching and Content Distribution Workshop*, Lisbonne, Portugal, mai 22–24, 2000.
  Parue aussi en *Computer Communications*, vol. 24, n. 2, pp. 207–214, février 2001.
- J. Kangasharju, J. W. Roberts, et K. W. Ross. Object replication strategies in content distribution networks. *Proceedings of 6th Web Caching and Content Distribution Workshop*, Boston, MA, juin 20–22, 2001.
  Parue aussi en *Computer Communications*, vol. 25, n. 4, pp. 376–383, mars 2002.
- J. Kangasharju, K. W. Ross, et D. A. Turner, Optimal Content Replication in P2P Communities, en soumission.
- J. Kangasharju, F. Hartanto, M. Reisslein, et K. W. Ross. Distributing layered encoded video through caches. *Proceedings of IEEE Infocom*, Anchorage, AK, avril 22–26, 2001.
  Une version approfondie acceptée pour publication dans *IEEE Transactions on Computers* special issue on QoS Issues in Internet Web Services en 2002.

Chapter 1

Introduction

In recent years the Internet and the World Wide Web have exploded in popularity. The Internet has become an important source of both information and entertainment for people all over the world. The phenomenal growth has been spurred on by the increasing ease of use of browsers, and the addition of more appealing content, such as audio and video. As people are getting faster access at home, through DSL and cable modems, the demand for high-quality content will increase. Also, as wireless devices, such as mobile phones and PDAs, are becoming more popular, people will be able to access the Internet from anywhere and at any time.

This ever-increasing demand for content places a high burden on the Internet infrastructure. The servers and network links must be able to accommodate the demand which can be highly variable and unpredictable. Some content, such as major portal sites like Yahoo [94], MSN [49], or Netscape [54], are constantly popular, and the site operator can take steps to increase the availability of the site. Balancing the load over several servers and installing higher capacity network links can greatly help in providing better and faster service to user requests.

The major problem is the so-called hot-spots. These happen when some content suddenly becomes extremely popular, typically for only some period of time. Examples of these hot-spots are major sporting events, other major news events, or releases of popular software. All these events generate a high load on the server originating this content, but it is hard to predict exactly how much capacity is needed. Sometimes the events are so sudden, e.g., someone publishing a link to a web page on a discussion board and members of the board all flock to that page, overloading the server and saturating the links. Handling these hot-spots is the major problem in Internet content distribution.

The basic solution, and on which all modern content distribution architectures are based, is to replicate the content on several servers and have different clients access different servers. This guarantees that the loads on the servers and network links stay manageable and the users are able to get the content they want in a reasonable time. This presents us with two problems. First, where and how to replicate the content, and second, how to make sure clients are able to find the replicated content.

The focus of this thesis is on content distribution architectures. This thesis is in two parts. In the first part we address client redirection methods and in the second part we address object replication strategies.

An efficient client redirection should achieve two goals. First, it should balance the load as evenly as possible on the set of available servers. Second, it should be easy to deploy in practice. This second requirement turns out to be far stricter of the two, because it typically requires that we cannot change the software on all clients. There is already a wide base of installed software and it is not possible to force everyone to install a new version of browser software to take advantage of the redirection features. This implies that the redirection mechanism should insert itself transparently somewhere in the existing request-response chain. An alternative approach is to use mechanisms which allow for incremental deployment, such as the Location Data System which we present in Chapter 3. The advantage of mechanisms which insert themselves in the request-response chain, such as DNS redirection as practiced by modern content distribution networks, is that they are immediately available to all clients. The disadvantage of these methods is that they are not necessarily fully transparent to the user, and the user may experience even a loss of performance.

Object replication mechanisms are closely tied with redirection mechanisms, after all, when the objects have been replicated to several servers, we need to be able to direct clients to these servers. Object replication is typically driven by the content provider or on behalf of the content provider, hence it is possible to use more elaborate mechanisms which may require changing even all of the existing server software. This gives us the possibility to coordinate the placement of replicas and condition it on the available server and network capacity, and prevailing client demands. Indeed, as we show in Chapter 6, coordinated placement of object replicas provides the best performance, compared to methods which try to act without coordination. Together with client redirection, object replication offers us the possibility to spread the load over several servers, and allows for efficient delivery of content to the users.

In Chapter 2 we present an overview of how modern content distribution technologies and architectures have evolved from the basic client-server architecture into the modern content distribution networks. We will pay particular attention to how these technologies have addressed the problems of redirection and replication – what are the advantages and disadvantages of these solutions.

In the following we outline the major contributions of this thesis.

1.1 Thesis Contributions

This thesis makes several contributions. The first contribution (Chapter 3) is an architecture for locating copies of objects. This architecture is based on extending the Domain Name System (DNS) and allows clients to locate origin servers, mirrors, or caches which hold a copy of the requested object. This architecture requires only minor extensions to DNS and allows for incremental deployment. We also present methods which clients can use to determine which of the available copies is "the best", i.e., can be downloaded the fastest.

The second contribution (Chapter 4) is a new, replicated architecture for an Internet-wide directory service. We also show how this architecture can be applied the Domain Name System. The key novel feature of this architecture is that it allows us to store rapidly changing information in the DNS. Currently, because of extensive use of caching, the rate of change in the DNS is severely limited. Our architecture leverages satellite and multicast communication technologies which, in addition to decreasing cost of hard disk space, allow us to effectively replicate the entire DNS database on selected servers. This provides us with the possibility of keeping rapidly changing information coherent in all servers, and also significantly decreases the DNS query latency. Our architecture is completely transparent to legacy DNS clients and can be deployed incrementally with no change to existing DNS software. We also perform an extensive performance analysis of our architecture.

The third contribution (Chapter 5) is an evaluation of the redirection techniques used in modern content distribution networks. Currently the content distribution networks use DNS to redirect clients to content servers. We can distinguish two types of redirection models – full and selective redirection. Using extensive simulations and experiments with live servers on the Internet, we compare how these two redirection methods affect the overall user-perceived performance. Our results indicate that full redirection yields superior performance because it avoids setting up new TCP connections which can severely degrade the performance of the selective redirection method. Although this result may seem intuitive, our contribution in this chapter is the quantitative, numerical evaluation of this tradeoff and its effects on user-perceived performance.

As our fourth contribution (Chapter 6), we study object replication strategies in content distribution networks. We formulate the problem of optimal placement of replicas as a combinatorial optimization problem which we show to be NP-complete. We develop several heuristics for determining the placements of the replicas and using real Internet topologies, we evaluate the performance of our heuristics. Our results indicate that the best performance is obtained when object replication is coordinated over the whole network; strategies which operate only locally achieve inferior performance. We also develop a cooperation model for peer-to-peer networks. Our evaluation shows that cooperation can yield significant benefits for the cooperating peers.

Our fifth contribution (Chapter 7) is a suite of algorithms for replicating and locating objects in a peer-to-peer community. We distinguish between two types of communities: content caches and content clubs. For both cases, we formulate the problem as an integer programming problem which provides us with the optimal solution. We provide several adaptive, distributed algorithms for replicating the objects on-the-fly and we evaluate their performance through extensive simulations. We find that by combining our algorithms with a distributed least-frequently-used replacement policy, we can achieve near-optimal performance, both in terms of hit-rate and number of replicas. We also consider a modified version of the integer programming problem which extends the initial problem to include constraints for balancing the load evenly over all peers.

As our sixth contribution (Chapter 8), we study the distribution of layered encoded video through caches. We consider the two-resource problem of cache space and network capacity and using a stochastic knapsack model we formulate the problem of layered video caching. In our model clients may request some number of layers from the video, depending on their network connectivity. Layered encoded video imposes us with the decoding constraint, i.e., all lower layers need to be present to be able to decode and show a given layer. We propose several heuristics for determining which layers of which videos should be cached in order to maximize the accrued revenue. We evaluate the performance of our heuristics through extensive numerical experiments. In addition, we also consider two intuitive extentions, quality negotiations and request queueing, but our results indicate that they have only a secondary effect on the overall performance. We also consider the case of partial caching in the context where clients are only interested in full-quality videos. We discover that in this case partial caching, i.e., caching only some lower layers of a video, is not beneficial and that in this case we should always cache complete videos.

1.2 Thesis Organization

This thesis consists of an introductory chapter which presents an overview of modern content distribution technologies, six research chapters in two parts, and a concluding chapter.

In the first part of this thesis, Chapters 3, 4, and 5, we focus on client redirection. Chapter 3 presents the Location Data System. It is a system designed to inform clients of copies of objects which exist in the network. Chapter 4 describes our architecture for a replicated Domain Name System. We present an overview of the architecture, a performance evaluation, migration strategies from the existing DNS, and consider security and fault tolerance issues. Chapter 5 evaluates the performance of the redirection schemes used by modern content distribution networks. Through extensive simulations and experiments on the Internet, we evaluate the effects of redirection on user-perceived performance.

In the second part of this thesis, Chapters 6, 7, and 8, we focus on object replication. Chapter 6 presents a model for optimal replication of objects in a content distribution network, develops several heuristics, and evaluates their performance using real Internet topologies. We also develop a cooperation model for peer-to-peer networks. In Chapter 7 we consider optimal content replication in peer-to-peer communities. We develop a model which provides us with the optimal solution, and we present adaptive algorithms that can be used to replicate content on-the-fly. Chapter 8 considers the problem of caching layered encoded video. We present a model based on stochastic knapsack, develop several heuristics, and evaluate their performance through extensive experiments.

Each of the six research chapters is self-contained and in addition to the main contribution, presents an overview of related work and future directions for that research. Finally, Chapter 9 summarizes the results and contributions of this thesis.

1.3 Published Work

Chapters 3, 4, 5, 6, 7, and 8 have been published, or are currently in submission:

- J. Kangasharju, K. W. Ross, and J. W. Roberts. Locating copies of objects using the domain name system. In *Proceedings of 4th Web Caching Workshop*, San Diego, CA, March 31 – April 2, 1999.

- J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 26–30 2000.

- J. Kangasharju, K. W. Ross, and J. W. Roberts. Performance evaluation of redirection schemes in content distribution networks. In *Proceedings of 5th Web Caching and Content Distribution Workshop*, Lisbon, Portugal, May 22–24, 2000.

  Also published in *Computer Communications*, vol. 24, n. 2, pp. 207–214, February 2001.

- J. Kangasharju, J. W. Roberts, and K. W. Ross. Object replication strategies in content distribution networks. In *Proceedings of 6th Web Caching and Content Distribution Workshop*, Boston, MA, June 20–22, 2001.

  Also published in *Computer Communications*, vol. 25, n. 4, pp. 376–383, March 2002.

- J. Kangasharju, K. W. Ross, and D. A. Turner, Optimal Content Replication in P2P Communities, in preparation.

- J. Kangasharju, F. Hartanto, M. Reisslein, and K. W. Ross. Distributing layered encoded video through caches. In *Proceedings of IEEE Infocom*, Anchorage, AK, April 22–26, 2001.

  Extended version accepted for publication in *IEEE Transactions on Computers* special issue on QoS Issues in Internet Web Services.

Chapter 2

Overview of Content Distribution

In this chapter we will give an overview of content distribution on the Internet. The main focus of this chapter, as well as this thesis, is on content distribution on the World Wide Web (WWW or the Web). We will present an overview of modern content distribution technologies and how they have evolved over the past few years. In addition to Web content distribution methods, we will also present a new content distribution paradigm, namely peer-to-peer content distribution.

In this thesis, the term *content* denotes any kind of content that is on the Internet and is available to the public at large. This could be for example Web pages, MP3-files, program downloads, etc. For the most part, we assume this content to be relatively stable, i.e., the content does not change much. If the content is being modified actively, this presents us with the problem of maintaining coherency of the content and assuring that users never see stale content. In Chapter 4 we consider how to maintain coherency in a replicated database when the information is changing rapidly.

We will use the term *content provider* to denote an entity (a company or an individual) who provides content for others to download. We use the term *user* to denote the persons who access this content. Typically content providers and users are different, but in a peer-to-peer network all users are also content providers.

2.1 World Wide Web

The World Wide Web has become one of the main channels for distributing information in the world today. Most major companies and organizations operate web sites and people often turn to the Web for finding information or entertainment. The Web is also widely used for commercial transactions, such as buying books, CDs, DVDs, or making travel reservations. These transactions typically also require efficient databases and secure transactions for purchasing the desired items. Yet, there is still the need to deliver Web pages to the users so that they can make their purchases.

The focus of this thesis is on distributing content from the content providers to the people who want to access this content. We focus our studies on how this content can most efficiently be delivered, either in the shortest time, or using the least amount of resources.

We do not address the problems of managing databases or conducting secure transactions in this thesis.

The Web is a client-server architecture. Content on the Web is hosted on *web servers* and is served using the HyperText Transfer Protocol (HTTP) [22]. Users use browser software to access the content stored on the servers. Objects in the Web are identified by Uniform Resource Locators, or URLs [8], which identify the name or the server which has the object, and also where on that server the object resides. Web pages are written in HyperText Markup Language (HTML) [93] and can contain text, images, and links to other web pages.

This makes the Web very easy to access for novice users and has been the main reason behind the popularity of the Web. The increasing popularity has also caused many problems in the Web. Some content is more popular than other content and hence the majority of requests are for the popular content. This puts a major load on both the servers serving the popular content and on the network links leading to these servers. Upgrading the servers or network links is not a feasible solution because millions of new users will want to access that same content.

A widely used solution is to replicate the contents of a server on another server, called *mirror server* or mirror for short. The content on the mirror is identical to that on the main server, so users can access either one and get the content they want. The problem is informing users of the existence of these mirrors. Typically the main server has a list of mirrors and the user must manually select one every time he visits the site (unless he stores the address of the mirror in a bookmark). Users are not always aware of the existence of these mirrors and may not be savvy enough to know why they should select one of them. Also, handling multiple mirros may be demanding for the administrator of a server because he needs to ensure that the content on the mirrors is up-to-date.

The problem of content distribution arises from the inefficiencies of the client-server architecture which is unable to handle the demands on the Web. In the following we will present how the technologies have evolved to combat this problem.

2.2 Proxy Caching

The first solution was to place a caching proxy[1] at the client site. This setup is show in Figure 2.1. In this setup the client sends its request to the caching proxy which checks if it has the requested object in its cache. If it does, it can return the object directly to the client and avoid contacting the server (also called origin server) which can be far away in the network. If the caching proxy does not have the requested object, it will download it from the origin server, deliver it to the client, and cache a copy of the object.

Typically there are several users behind a single caching proxy, for example, an institution (university or company) or an Internet Service Provider (ISP) would install one proxy for all of its users who would access the Web through this proxy. This creates a large user

[1] The terms proxy cache, caching proxy, and proxy are used interchangeably in this thesis for a caching proxy [14]

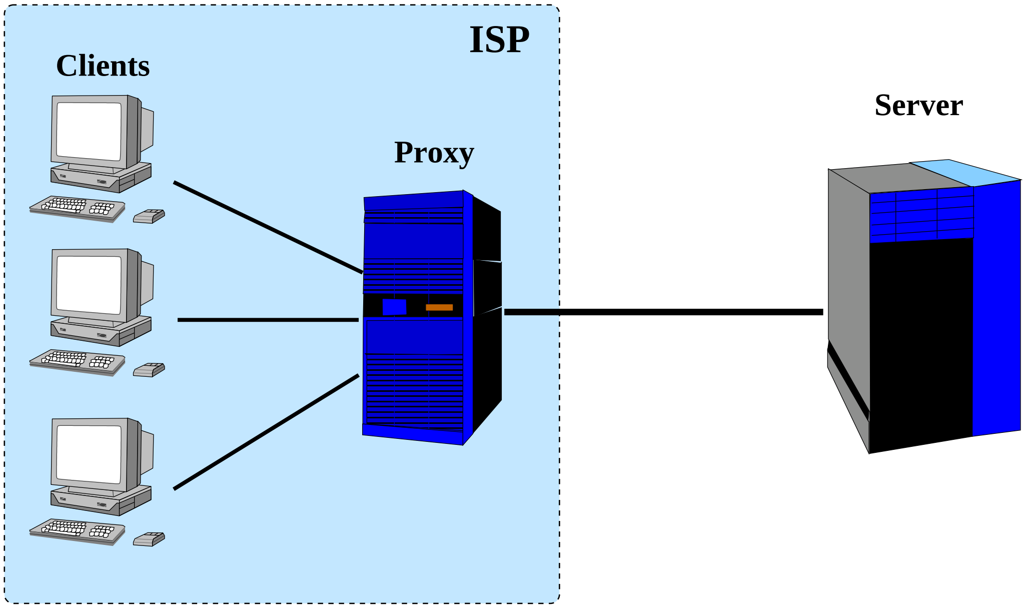


Figure 2.1: Client, Proxy, and Server

community and because popular content tends to be accessed by many people, the cache on the proxy is able to satisfy many requests and reduce the download times for the users, reduce bandwidth usage for the institution, and reduce load on the origin server.

A *cache hit* is a request from a client that the proxy served from its cache. A *cache miss* is a request which the proxy was not able to serve from the cache. Cache misses can occur either because the requested object simply was not in the cache, or because the cached copy was not valid anymore. An important metric for evaluating the performance of a caching proxy is the hit-rate (also known as hit-ratio or hit probability). Hit-rate is defined as

$$HR = \frac{\text{\# of requests served from cache}}{\text{total number of requests}}. \tag{2.1}$$

In a similar manner, we can define the byte hit-rate which is the ratio of bytes served from the cache as cache hits to the total number of bytes that have passed through the cache. Because the object sizes on the Web vary significantly, maximizing hit-rate does not typically maximize byte hit-rate and vice versa.

Because the cache is of a limited size, we can only store a subset of the objects seen by the cache. When the cache is full and we want to store a new object, we need to evict one or more objects from the cache. The *replacement policy* of the cache determines which objects are evicted. There has been a considerable amount of research in Web cache replacement policies, see [33, 92] and references therein. In practice, the most widely used replacement policy is the least-recently-used (LRU) replacement policy which is also widely used in virtual memory systems in modern operating systems [84]. LRU is also the default policy in the freely available Squid caching proxy [81]. Even though other replacement policies in some cases achieve higher hit-rates, LRU is generally considered to perform sufficiently well.

Using caching proxies the popular content is always automatically stored (or replicated) near where it is needed. Because caches may evict objects, this is different from mirrors which typically are permanent and stable copies. To use the caching proxy, users need to configure their browser software, or the administrator of the institution can install an intercepting proxy[2] [14] which intercepts all user requests and directs them to the cache. Hence, the redirection to the cache may not happen always and users typically have the possibility of forcing the cache to load the page from the origin server.

2.2.1 Caching Hierarchies

Even though caching proxies are useful and efficient for pooling the users of a single ISP or institution, they can cover only a relatively small user population. Consider for example users in a European country wanting to access content in the US. Each user goes through his local caching proxy, but if the requested object is not there, the proxy will contact the origin server in the US. This means the for every miss, each proxy must go over the expensive transatlantic link to retrieve the content.

Caching hierarchies attempt to alleviate this problem by having the institutional proxies contact other proxies before contacting the origin server. Figure 2.2 shows an example of a caching hierarchy. In Figure 2.2 we have four caches I_1, I_2, I_3, and I_4 at the institutional level, for example in universities or dial-up ISPs. The institutional caches connect to regional caches R_1 and R_2 which, in turn, connect to the national cache N. (The hierarchy in Figure 2.2 has three levels, institutional, regional, and national, but in reality there may be more or less levels.) Such nation-wide caching hierarchies have been deployed in several countries, for example in France [68] and in the US [52].

Client requests in caching hierarchies proceed as follows. Suppose a client under the institutional cache I_1 wants to request an object which resides on some origin server. First, the client sends its request to I_1. If I_1 has the object cached, it will return it to the client (after possibly verifying that the cached copy is still valid). If I_1 does not have a copy, it will send the request to R_1. If R_1 has a copy, it will return it to I_1 which will send it to the client *and* cache a copy locally. If R_1 does not have a copy, it will send the request to the national cache N. If there is a miss at N, the national cache will retrieve the object from the origin server, cache a copy, and pass it down to R_1. As objects are requested through the hierarchy, copies are automatically created along the path. If another client under I_1 requests the same object, I_1 is able to satisfy the request from its cache. Likewise, if a client under I_2 requests the same object, the request will only have to traverse up to R_1 which has a copy of the object.

In addition to having the proxies form up a strict hierarchy, they can also cooperate

[2] The term transparent proxy is also widely used because the user does not need to configure the browser. From a networking point of view, these proxies are not transparent because they can change the semantics of the requests.

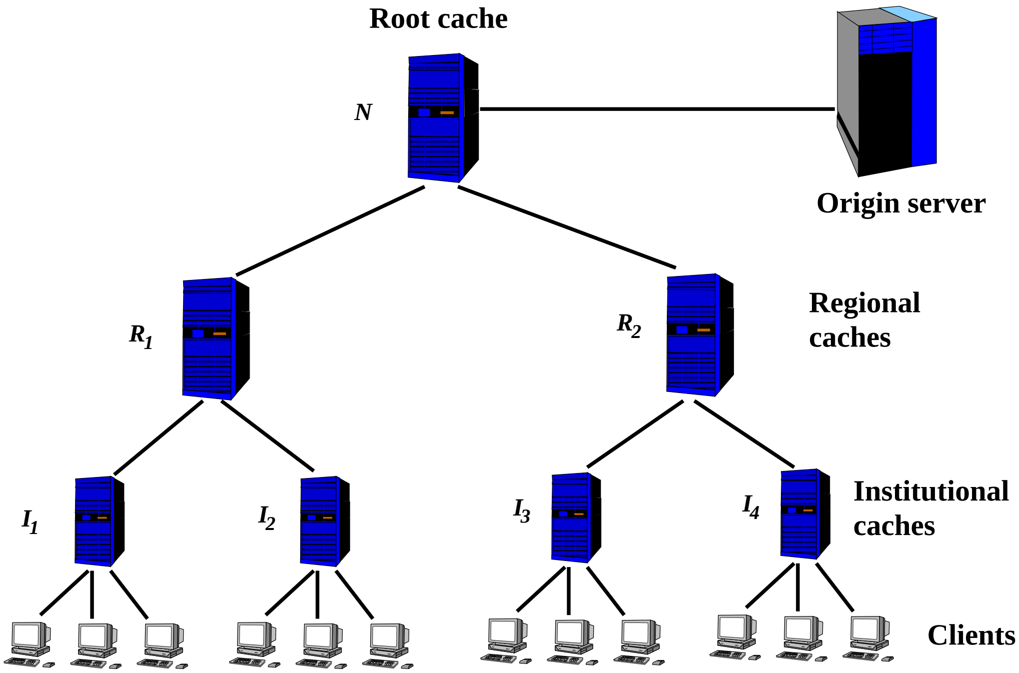


Figure 2.2: Caching hierarchy

in other ways. For example proxies I_3 and I_4 in Figure 2.2 could form a group and every time a either of the two has a miss, it would query the other proxy for the object. In this manner, if any of the group members has the object, the requesting proxy would retrieve the object from it, instead of contacting the origin server. There are several protocols, such as Internet Cache Protocol (ICP) [91] and Cache Digests [73], that have been developed for cache cooperation.

Caching hierarchies automatically replicate the most popular content near the users and the moderately popular content higher up in the hierarchy. Caching hierarchies can be seen as a form of asynchronous multicast [70] because, ideally, each objects traverses a given link only once, although object replacement in practice may affect this. Redirection happens as with the institutional caching proxies, that is, either users must configure their browsers or we have to use intercepting proxies. To build the caching hierarchy, the cache operators at different levels have to agree on who can access content through their caches. Because such hierarchies have typically been funded through research organizations, use of these hierarchies has widely been allowed.

One main problem with a caching hierarchy is that it may introduce a significant additional latency to requests. If a client requests an unpopular object which is not cached in any of the caches, the request will still have to traverse all the way up in the hierarchy before the root cache sends the request to the origin server. This latency may be increased by protocols such as ICP which, in the worst case, may impose an additional 2 second delay at each step in the hierarchy. Also, there are no guarantees that the caches higher up in the hierarchy are any closer to the origin server. This is typically handled by filtering which requests are sent up in the hierarchy. For example, the national cache would only see requests for objects that are located (as indicated by the URL) in other countries.

2.2.2 Problems with Client-side Caching

Installing caching proxies on the client-side, whether a simple institutional proxies or a caching hierarchy, has not been without problems, however. The main problem with this approach is that the content provider has no control over the content once it has been retrieved from the origin server and placed into the caches. The content provider has no good way of ensuring that if the content changes, then all users would always get the new content. There are some simple mechanisms to inform caches about the content, such as the `Expires`-header or cache control mechanisms of HTTP/1.1, but these have never been widely deployed.

In the absence of reliable information about the object, caching proxies have resorted to heuristics to determine how long they can consider the cached object valid. One such heuristic considers that if the object has not been modified in a long time, it is unlikely to be modified in the near future and hence it can be cached for a long time. This is the heuristic programmed into the freely available Squid proxy cache [81].

In order to avoid the problem of caches delivering stale, or out-of-date content, some content providers have resorted to marking *all of their content* as non-cachable so that the caching proxies would not store it. This solution, while effective at solving the problem of stale content, defeats the purpose of installing caches because they cannot store content anymore. This problem was the primary motivation behind the development and success of *content distribution networks* (CDN) which we will present in the next section.

2.3 Content Distribution Networks

In early 1999 and in the years since, several companies have started to operate their content distribution networks or CDNs. These companies include Adero, Akamai, Digital Island, Mirror Image, and SandPiper [2, 3, 16, 43, 76]. This new model of content distribution has become very popular among the large content providers.

Figure 2.3 shows the architecture of a CDN. A CDN makes agreements with the content providers (O_1, O_2, and O_3) to distribute their content to the users. The CDN operates content servers (C) that are typically placed near the users, for example at the dial-up ISPs. The user requests are redirected to these content servers which are able to serve the content fast. The internal network of the CDN connects the origin servers and content servers and is used to transfer content from the origin servers to content servers (or moving content from one content server to another).

The main difficulty in creating a CDN is redirecting the clients to the content servers. Ideally, one would want this to be completely transparent to the clients so that no modifications to client software is needed. Modern CDNs achieve this by manipulating information in the Domain Name System (DNS) and in the following we present the two methods currently in use. Chapter 5 presents a detailed study of the performance of these two

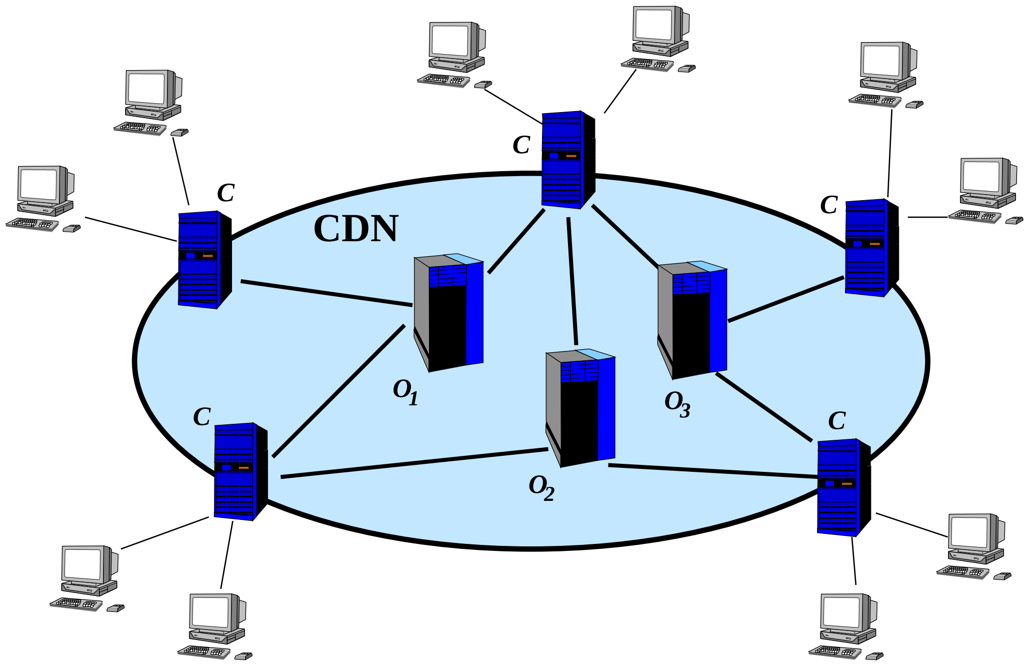


Figure 2.3: Content distribution network

schemes.

Because CDNs charge a high price for their services, they are mostly applicable to larger companies wanting to distribute their content. Individuals or small organizations would typically not be able to afford the services of a CDN, hence they would have to rely on client-side caching to help distribute their content. CDNs are therefore not a replacement to the traditional client-server model or client-side caching, but a complementary approach which allows for efficient delivery of very popular content to a large number of interested users.

2.3.1 Full Replication

In the *full replication* scheme the CDN takes control of the DNS mapping of the content provider's server, say `www.example.com`. When a client wants to request an object from this server, it first has to do a DNS lookup on `www.example.com` to get the server's IP address. The information in the DNS system for the domain `example.com` points to a nameserver in the CDN's network. When this nameserver receives the client's DNS lookup request, it determines which content server is the best placed to handle this request and it returns the IP address of that content server as the IP address of `www.example.com`. When the client receives the DNS reply, it will attempt to connect to the content server. Because the content server is closer to the client than the origin server, the client will receive the requested object much faster.

The downside of this approach is that each content server must be able to handle all requests for the content provider's origin server. This implies that either each content server fully replicates the contents of all content providers with which the CDN has passed an agreement, or that the content server acts as a surrogate proxy [14].

The benefit of this mechanism is that *all client requests* are always sent to the content servers. We will study this issue deeper in Chapter 5.

2.3.2 Partial Replication

In the *partial replication* scheme, only a subset of the objects on the content provider's origin server are placed on the content servers. Client redirection goes as follows. The client retrieves the home page from the origin server `www.example.com`. This page may contain references to images which have been placed on the content servers. The URLs of these images have been changed, for example the URL `http://www.example.com/title.gif` could become `http://www.cdn.net/example/title.gif`. From the client's point of view, this image looks like any normal image, except that it has to open a connection to a new server. When it opens this connection, it has to perform a DNS lookup to get the IP address and this DNS lookup is handled in the same way as above. The client gets redirected to a nearby content server and asks it for the image.

The benefit of this mechanism is that only objects that need to be replicated are placed on the content servers, thus reducing the storage requirements. But the downside of this mechanism is that someone, typically the content provider, has to decide which objects are to be replicated. This means that the system as a whole is slow to react to hot-spots which may occur when some content on the origin server suddenly becomes extremely popular. Also, the cost associated with opening a new connection to the content server is non-negligible, as we show in Chapter 5.

One weakness that is shared by both replication mechanisms is that the redirection decisions are based on the IP address of the machine which sent the DNS lookup request. This is typically the nameserver for the client and may or may not be topologically close to the client. The effectiveness of DNS-based server selection has been studied in more detail in [79]. Their results indicate that especially for dial-up users, the cost of DNS redirection can be very high.

2.4 Peer-to-Peer Networks

The latest development in content distribution is peer-to-peer networks. The first peer-to-peer network was Napster [51] which allowed users to share MP3-files with each other. The main application for peer-to-peer networks has been file sharing, in which users make some files available on their computers and others can download these files. In order for users to be able to find out which users are offering which content, the network needs some kind of a lookup service which maps object names into the machines serving these files. Below we will discuss some possible approaches for building such a lookup service.

What sets peer-to-peer networks apart from the traditional forms of content distribution, caching and CDNs, is that in a peer-to-peer network every node is both a client and a server. However, studies have shown that in reality, most of the content in a peer-to-peer network is served by a small minority of users and a large number of users do not offer any files [1]. Regardless of this, peer-to-peer networks have become extremely popular for sharing files between users.

The main problem faced by Napster was not technical, but legal. Napster was made to share only music encoded in MP3-format and most users used it to share and retrieve copyrighted music without permission from the copyright owners. This prompted a long legal battle between Napster and record companies and as a result Napster was forced to shut down[3].

Regardless of this problem, Napster had showed the advantages of the peer-to-peer content distribution model and several new systems and protocols have been developed, for example, Gnutella [28], Freenet [23], FastTrack and Morpheus [21,48], and MojoNation [47]. There has also been considerable interest in the research community shown by the large number of peer-to-peer projects, such as CAN [63], Chord [83], Pastry [75], Tapestry [95], OceanStore [39], and FarSite [20]. We will review these projects in detail in Chapter 7.

We will now present the main approaches for building a lookup service for a peer-to-peer network, namely centralized, distributed, and hybrid.

2.4.1 Centralized Architecture

Figure 2.4 shows an example of a centralized peer-to-peer architecture. The centralized architecture requires that some central authority operates a single central server. (This server would typically be a server farm, but the users would see it as a single server.) This central server is responsible for answering the queries, hence all the query traffic is directed to it.

The lookup service in Napster was based on a centralized architecture. When a user wanted to locate an object, the Napster software would contact a server, operated by Napster, that kept track of who was online and which files each user was sharing. The central server would perform a simple keyword search with the search terms given by the user, and would return a list of potential matches. The user would see this list and would have to choose one peer from whom to download the file. The list would include hints, such as other peers' network bandwidth and round-trip times, to help users choose peers which are close-by or well connected. However, these hints were not verified, and any user could indicate any connection bandwidth he desired. The actual file transfer would happen directly between the peers and not via the central server.

The main drawback of this architecture is that the central server becomes a potential bottleneck and is a single-point of failure. Using a server farm we can overcome the bottleneck problem, but if all the servers are co-located, a single network failure could take out all the servers. The servers can be geographically dispersed and peers redirected on them based on some criteria, but in this case the quality of service obtained by the peers depends on which server they happened to be redirected. In this model, each peer would have to register with the central server when the software was installed, and also to notify the central server when the peers goes up or down, and which files it has.

[3] Napster has recently started its services again, using a subscription based model.

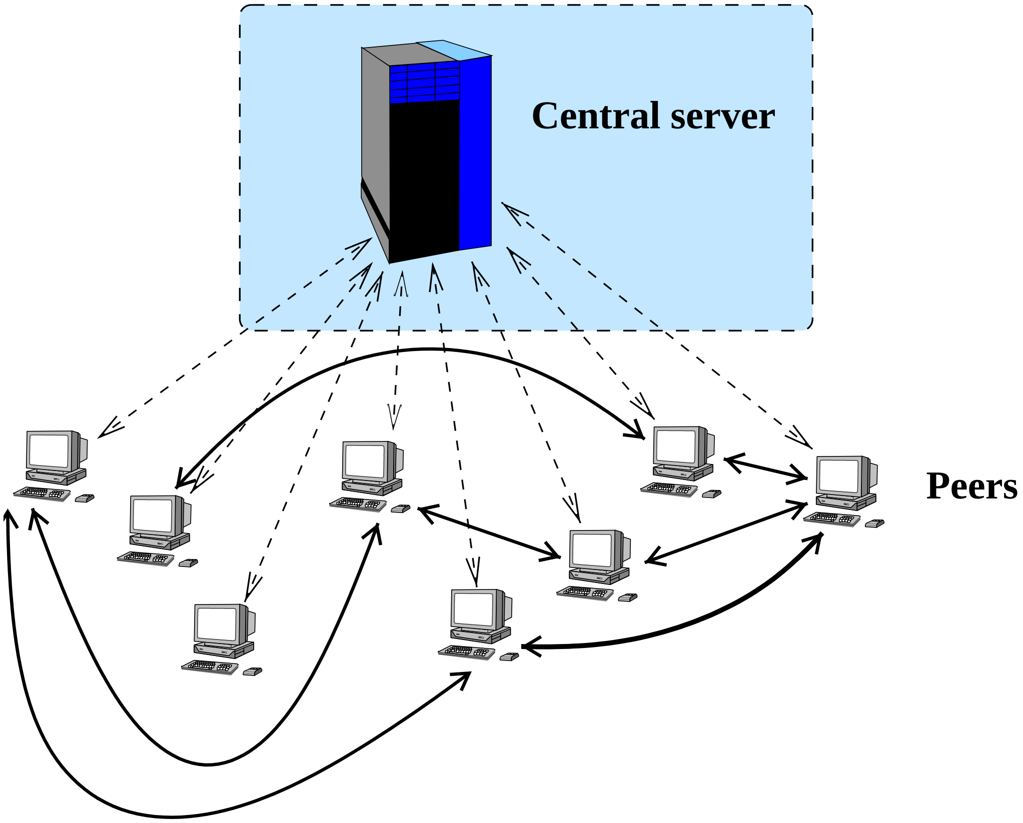


Figure 2.4: Centralized peer-to-peer network. Dashed lines show query traffic and solid lines show how objects are transferred.

The advantage of this model is that the queries are simple to perform and they do not consume much network resources. Each client request results in one query message, from peer to central server, and one longer response, the list from the central server to the requesting peer.

2.4.2 Distributed Architecture

The other main lookup architecture used in peer-to-peer networks is a distributed architecture, such as the one used in Gnutella [28]. Figure 2.5 shows a distributed peer-to-peer network. The main advantage of a distributed architecture is that all peers are equal and no peers hold any permanent information about which objects are stored where; also there is no directory of the peers which are a part of the network.

When a user wants to join such a network, he must typically first obtain the IP address (or hostname) of a peer that is already a member of the network. This could be achieved through some well-known bootstrap nodes and the peers can obtain their addresses either from the DNS (the approach in CAN [63]) or simply with out-of-band methods, such as publishing the addresses of bootstrap nodes on a web page. Note that there can be one or more such bootstrap nodes. Also, as is the case in Gnutella, each peer can also serve as a

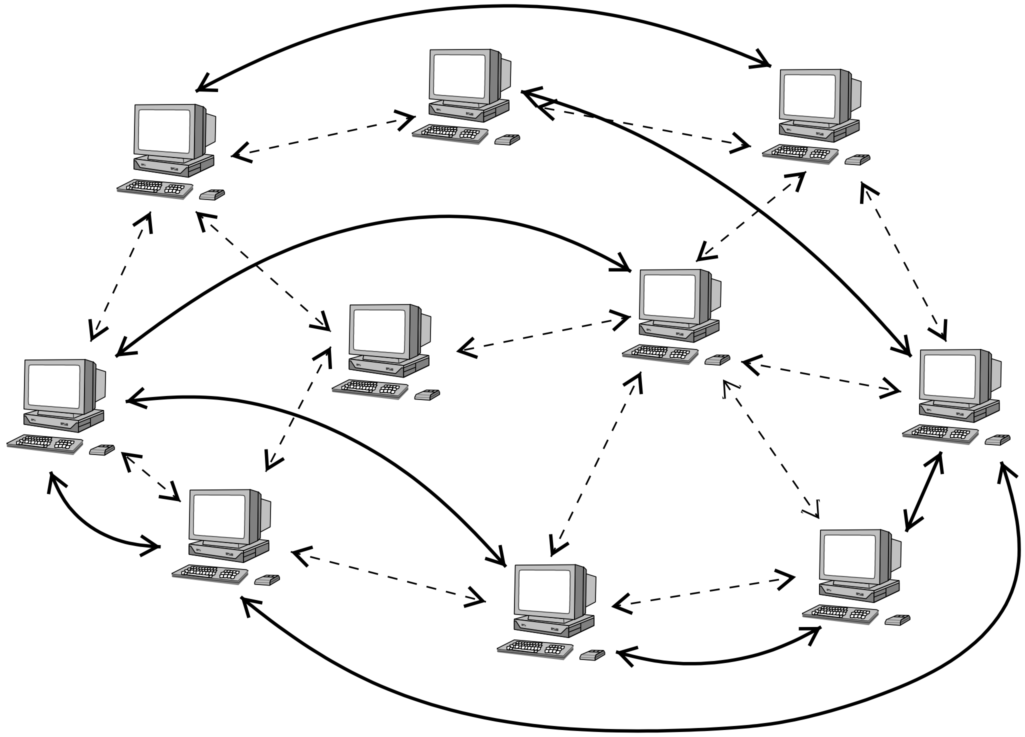


Figure 2.5: Distributed peer-to-peer network. Dashed lines show query traffic and solid lines show how objects are transferred.

bootstrap node, but this still requires that the new peer is able to obtain the address of at least one peer in the network.

The first peer-to-peer network to use the distributed architecture was Gnutella and when a Gnutella-peer wants to retrieve an object, it queries as follows. The requesting peer has some number of neighboring peers in the peer-to-peer network[4]. It sends its query to all of its neighbors. If a neighbor has the requested object, it will reply to the requesting peer and inform it that it has a copy. If a neighbor does not have a copy, it will, in turn, send the same query to all of its neighbors, excluding the neighbor who sent the original query.

This way the query eventually propagates to all peers in the network and the requesting peer is able to find out if any peer has a copy of the object. This also means that every query needs to be flooded to all of the network which puts a considerable strain on it. Gnutella attempts to alleviate this flooding problem by using a limit (time-to-live, or TTL) on how many times a query can be forwarded. This TTL is set by the requesting peer, hence a user can increase it if a query did not find the object. If some user sets this TTL very high, he could receive potentially thousands of replies which could clog up his network connection. Hence, users will learn, through trial-and-error, what is the best setting for this TTL.

[4] Neighbor relations in the peer-to-peer overlay network do not imply that the two peers are near each other in the actual network topology.

Because the query might not propagate to all peers, it is possible that a request ends in a failure, even though the object is available from some peers. In the centralized architecture this is not possible, because the central server is aware of all objects in the network.

Most of the current research on peer-to-peer networks has been aimed at improving query performance in distributed peer-to-peer networks [63, 75, 83]. We will review these projects in more detail in Chapter 7.

2.4.3 Hybrid Architectures

FastTrack [21] and Morpheus [48] use a hybrid architecture for querying. This hybrid architecture attempts to strike a balance between the accuracy of the centralized architecture and the lower load of the distributed architecture. In the Morpheus architecture, some peers have been designated as supernodes by the bootstrap node. Normal peers are assigned a supernode by the bootstrap node. When a peer wants to request an object, it sends its query to its supernode. Each supernode maintains a directory of all the objects in the peers under it; this provides for Napster-like behavior within the peers under a supernode. The supernode can also forward the query to other supernodes which reply directly to the requesting peer and send the addresses of peers under them which have the requested object. This provides for a wide coverage, like Gnutella, but with considerably less resources.

There have not yet been any studies that would have compared the performance of the hybrid architecture with that of the centralized or distributed architectures.

2.5 Conclusion

This chapter has presented an overview of how content distribution technologies have evolved on the Internet in the past few years. Starting off with the basic client-server model, the first step was client-side caching. This was initially implemented with caching proxies, installed locally at ISPs or institutions, and later on these caches were used to create caching hierarchies. Client-side caching did not, however, allow the content provider any control over how the content would be cached. Content distribution networks emerged to remedy this problem and they have become the de-facto content distribution method for most commercial web content. Finally, we also presented a new content distribution paradigm, namely peer-to-peer networks. These networks differ from the traditional model in that each peer in the network is both a client and a server. We also presented different architectures for building a lookup service for a peer-to-peer network.

Part I

Client Redirection

Chapter 3

Locating Copies of Objects

3.1 Overview

In order to reduce average delay and bandwidth usage in the Web, geographically dispersed servers often store copies of popular objects. For example, with network caching, the origin server stores a master copy of the object and geographically dispersed cache servers pull and store copies of the object. With site replication, objects stored at master are replicated into secondary sites. In this chapter we propose a new network application, Location Data System (LDS), that allows an arbitrary host to obtain the IP addresses of the servers that store a specified URL. Our networking application is an extension to the Domain Name System (DNS), requires only small changes to the domain name servers, and can be deployed incrementally. For the case of network Web caching, we elaborate on our proposal to allow a cache to (i) update a distributed database when it stores or evicts objects, and (ii) push objects to parent caches in order to improve delay and bandwidth usage. For the case of mirrored servers, we show how a client can obtain a list of all servers mirroring all or part of the desired site. LDS applied to partially mirrored sites generates substantially less DNS traffic than LDS applied to caching. Finally, we discuss how a host can use the location data in order to make intelligent decisions about where to retrieve desired objects.

3.2 Introduction

Network caching of documents has become a standard way of reducing network traffic and latency in the Web. Caches are currently employed in institutional, local, regional and national ISPs. Cache hierarchies, created when caches in lower-level ISPs point to caches in higher-level ISPs, are currently prevalent in the Internet [52, 68]. Today's cache hierarchies use static, manually configured pointers to define the hierarchy tree. Cache hierarchies operate as follows. When a browser requests a document, it sends a request to a leaf cache. This cache then either serves the document (if it is cached) or forwards the document to its parent in the hierarchy. The process is repeated along a static chain of caches until the root of the hierarchy is reached. If there is also a cache miss at the root, the root forwards the request directly to the origin server. A response is returned along the cache chain in the reverse direction. Cooperating caches in cache hierarchies often use ICP (Internet Cache Protocol) to improve and enlarge the scope of the search [91].

Caching hierarchies have several problems. First, requests for less popular documents will experience misses at all caches in the cache chain. For deep hierarchies, these misses lead to poor latency performance [85]; moreover, ICP can further degrade performance, since the cache must wait for a reply from all sibling and parent caches or until a two second timeout before proceeding up the hierarchy. Second, today's caching hierarchies are static — they do not permit a browser or a cache to choose the subsequent cache according to current topology or traffic conditions. Manual optimizations, such as sending requests for certain top level domains to designated parent caches, are possible, but even with these optimizations, the hierarchy for a given URL remains static. Third, caching hierarchies do not allow the chain of caches to extend beyond the root cache; if there is a miss at the root server, the request is forwarded directly to the origin server, and never to a cache that lies somewhere in between the root server and the origin server. This is a problem because a cache nearby the origin might be able to serve an object much faster than the origin server, especially when the origin server runs on a slow machine or has a low bandwidth connection.

In this chapter we propose a new cooperative caching scheme that has the following features. (1) At most two servers (including the origin server) are visited in the request chain; (2) The chain of caches depends on the requested URL and can change dynamically as a function of current network topology and traffic conditions; (3) An arbitrary cache in the Internet can be queried, including a cache that is far from the browser but close to the origin server. (4) The scheme can be incrementally deployed with minor changes to DNS servers. Furthermore, although a thorough performance study is still required, we feel the scheme should lead to a substantial reduction in delay and network traffic as compared to traditional hierarchical caching.

Our caching scheme makes use of a new network application, the Location Data System (LDS), which we also define in this chapter. The LDS, defined as an extension of DNS, allows an arbitrary host to obtain the IP addresses of the servers that store a specified URL. The LDS is of independent interest, and can be used for other applications, including choosing the best mirrored site for a given URL. For the case of network Web caching, we specify how a cache updates the LDS distributed database when it stores and evicts objects, and how a cache pushes objects to parent caches in order to improve delay and bandwidth usage.

We recognize that LDS applied to Web caching significantly increases the number of DNS messages. In this chapter we also show how LDS can be applied to replicated and partially replicated servers. In this case, the amount of DNS messages does not increase.

This chapter is organized as follows. In Section 3.3 we define the LDS. In particular, we show how DNS can be extended to provide the LDS service. In Section 3.4 we show how network caches can exploit the LDS; in particular, we discuss how the caches update the LDS distributed database, and how caches at higher levels in the caching hierarchy can be populated. In Section 3.5 we show how document and site replication can exploit the LDS. In Section 3.6 we discuss how a host can make routing decisions based on the results of an LDS query. In Section 3.7 we discuss related research on cooperative caching. Section 3.8 presents directions for future work and Section 3.9 concludes the chapter.

3.3 Location Data System (LDS)

Copies of an object, with each object referenced by the same URL, are often available from different servers in the Internet. These servers include **origin servers**, **replicated servers** (also called mirrored sites) and **cache servers** (also called proxy servers). The origin server for an object is the server at which the object originates; it always contains an up-to-date copy of the object. A replicated server contains copies of objects that have been placed into it (typically manually or by pulling); typically, objects in replicated servers are up-to-date. A replicated server may replicate entire Web sites or may replicate only portions of various Web sites. Cache servers obtain copies of objects by pulling them on demand. In particular, when a cache server receives a request for an object, and if the object is not cached, the cache server retrieves the object from another server (which may be the object's origin server, a replicated server, or another cache server), stores a copy of the object, and forwards a copy to the requestor. A cached copy of an object may not be fully up-to-date. In this chapter we refer to all three types of servers as **object servers**.

It is highly desirable for a host in the Internet to be able to determine the locations (i.e., the IP addresses) of all the object servers that contain copies of a specified URL. In particular, a host would like to be able to give a network application a URL and receive from the application a list of all the object servers that contain the URL. In addition to receiving the IP addresses of all the servers that contain the object, it is desirable to receive information about the freshness of each copy, e.g., when each copy was last modified or the "age" of the object (as defined in the HTTP/1.1 [22]).

In this section we outline a new networking application that provides the service of mapping a URL to a list of object servers that contain the URL, with each server on the list having associated freshness information. We refer to this system as the **Location Data System (LDS)**. We have taken a rather pragmatic approach in designing the LDS. Our principle goal has been to design a system that can be rapidly deployed with incremental changes to the existing Internet infrastructure. A second goal is scalability, i.e., a system that is decentralized and hierarchical. A third goal is performance, that is, a system that quickly returns the location data while injecting a minimum of overhead traffic into the network.

Figure 3.1 shows the basic mapping service provided by LDS. In the figure, the client on the left has a URL, it gives it to the LDS system which returns a list of object servers that contain the object. Instead of the whole URL, the client can as well give only a prefix of the URL to the LDS black box which then returns a list of object servers that contain URLs matching the prefix. In the simplest case, the prefix is only the hostname of the URL; in this case, the LDS service is identical to the Domain Name System, operational in the Internet for over 20 years.

Given the similarities in the service provided by both DNS and LDS, we have decided to base the design of LDS on DNS. In fact, LDS can be implemented by making minor

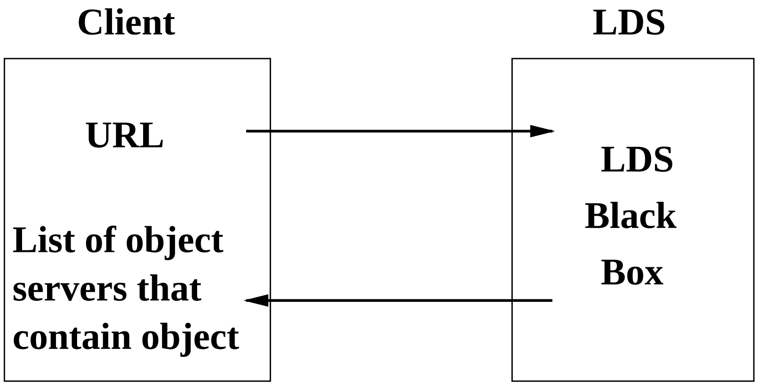


Figure 3.1: LDS service

extensions to BIND name servers and to the DNS protocol. Recall that DNS provides a mapping of hostnames to IP addresses. (DNS can return more than one IP address for a hostname for mirrored sites.) DNS uses a hierarchy of name servers to implement a distributed database of resource records, whereby a resource record contains a mapping.

We now present an overview of the LDS design; we provide more details in the subsequent subsections. As does DNS, LDS uses a hierarchy of servers to implement a distributed database of resource records. We refer to the LDS servers as **location servers**. Each LDS resource record maps a URL to an object server, with associated freshness information. Each URL has one (or more) authoritative location servers (To convey the main idea, we initially assume that each URL has exactly one authoritative server.) The authoritative location server contains a list of resource records for the URL, that is, a list of object servers that contain the URL (along with the freshness information). Other location servers may contain cached copies of the list of resource records. In our basic design, hosts query location servers in a manner that is fully analogous to the DNS protocol. The sequence of query and reply events is illustrated in Figure 3.2.

In Figure 3.2, C is the querying host (e.g., a browser), O_0 is the origin server for the queried object, machines O_1–O_4 are other object servers (e.g., caches), and machines L_1–L_4 are location servers.

The querying host sends a location query to its local location server (L_1). If L_1 does not have the location information cached, it sends a query to a **root location server** (L_2). If L_2 does not have the location information cached, it returns the address of a location server responsible for the domain of the origin server in the query (L_3). L_1 sends a new location query to L_3 and if L_3 does not have the location information, it returns the address of the authoritative location server (L_4). Finally, L_1 queries the authoritative location server and receives the location information. Location server L_1 then sends the information to the querying host and also caches the information. (In this example we assumed that all queries between LDS servers are iterative; recursive and combinations of recursive and iterative can also be used. Like DNS, LDS is not based on either query type being used. We also assumed that there is one intermediate location server between the

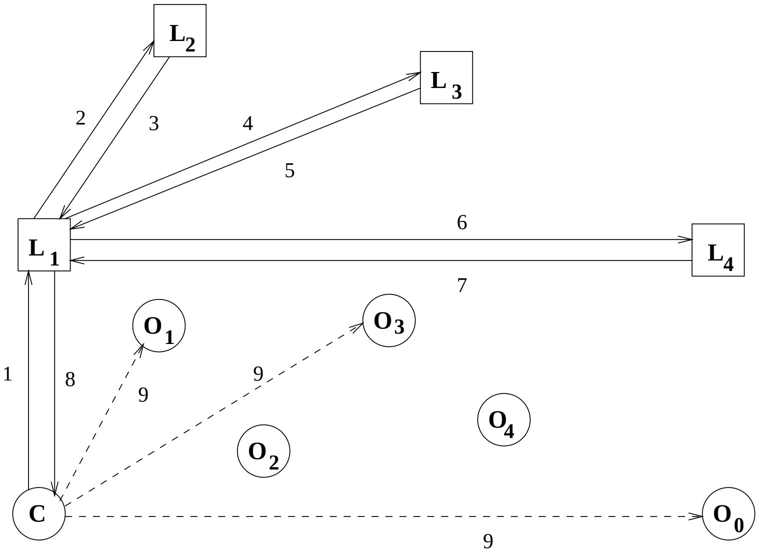


Figure 3.2: Sequence of Location Queries and Replies

4	**h**	**t**	**m**	**l**	**5**	**i**	**n**	**d**	**e**	**x**	**4**	**~**	**b**	**o**	**b**	**4**	**h**	**t**
t	**p**	**3**	**w**	**w**	**w**	**7**	**e**	**u**	**r**	**e**	**c**	**o**	**m**	**2**	**f**	**r**	**0**	

Figure 3.3: Representation of a URL

root and authoritative servers; in practice there can more or less.)

Suppose that the reply indicated that object servers O_0, O_1, and O_3 contain copies of the URL. The querying host then chooses to retrieve the URL from one of these object servers. Ideally, the host chooses the object server that will deliver the object the fastest. (We will discuss how this choice might be made in Section 3.6.)

Given the similarities between our basic design of the LDS and the DNS, we now address whether the LDS can implemented in the existing DNS, or within a slightly extended DNS. A DNS implementation could lead to rapid deployment of the LDS.

3.3.1 Resource Records, Query Messages and Reply Messages

For location requests a new DNS resource record type is needed. It is similar to the standard A-type resource record, which maps a hostname to an IP address. We now describe this new DNS resource record type.

Each resource record of this type must contain an object identifier. In order to be compatible with standard DNS queries, object identifiers must be encoded in a standardized way. Using this encoding, the query appears like a normal DNS A-query and can be resolved without changing any parsing routines in DNS servers. Figure 3.3 shows how the URL `http://www.eurecom.fr/~bob/index.html` would be encoded.

This encoding allows for efficient compression of identifiers sharing a common part, just as when querying the addresses of all of the servers in a given domain. Since the identifier can be viewed as a generalized hostname, all the normal facilities of DNS can be used.

We mention that there is a 255 character limit on hostnames in DNS queries which implies that URLs longer than 255 characters will have to be truncated. Also, the length of a single label (e.g., one component of hostname or URL path) is limited to 63 characters. We do not expect these limits to be a problem. We also mention that URLs are case-sensitive but DNS does not guarantee case-sensitive treatment. Nonetheless, we do not expect this to be a major issue since most URLs cannot be confused with other URLs even if case is not preserved.

We studied the access log from one of the NLANR top-level caches and out of the 1.1 million URLs in the file, only 122 exceeded these limits. Even if these 122 URLs were truncated to conform to DNS limitations, we did not observe any collisions between two different URLs. Neither did we find any URLs where case-insensitivity would have presented any problems. Therefore, we do not propose any methods for handling such URLs beyond simple truncation.

The reply will include several resource records, one for each object server holding a copy of the URL. Each of these resource records has a time-to-live (TTL) field which specifies how long that resource record can be cached at a DNS server. This TTL-information can either be specified by the authoritative location server or, for a better estimate, by the object server. If the location server sets this TTL-field, it will be the same for all resource records from that server. If, on the other hand, this field is set by the object server, it allows the object server to communicate information on how long the object is likely to be available at that server.

The actual data section of the resource record (RDATA) contains the IP-address of the object server and some freshness information about the object. This information could be for example the last modification date of the object, which provides an easy way of distinguishing between possible stale copies of the object. In this case, the authoritative location server would have to periodically query the origin server to get an up-to-date modification date for the original object.

Another possible piece of information that could be included in the resource record, instead of the last modification date, is the "age" of the object, as defined in HTTP/1.1. An object server could determine this locally, and a small age would refer to a relatively fresh copy. This method would not, however, offer the same guarantees on object freshness as would the last modification date.

Yet another possibility is to include simple TTL-information, but this would be redundant, since the resource record by definition includes a TTL-field. A TTL-estimate is useful when the querying host decides which object server to use. Object servers with short expected TTLs can be discarded from the decision making process.

Because all location servers are allowed to cache LDS replies, some location servers may have stale location information in their caches. When the status of an object changes at an object server, the object server notifies the authoritative location server. This update is not, however, reflected on the cached copies of the location information. If stale, cached location information is delivered to a querying host, the querying host may decide to forward the request to a Web cache that has evicted the object.

One solution is to use the `only-if-cached` directive of HTTP/1.1 when requesting objects from Web caches. If the distant cache has the object cached, it will return the object from the cache; otherwise it will return an error to the querying host indicating that the object is no longer cached. The querying host will then choose another object server from the list. We are currently looking into ways of reducing the amount of stale location information in LDS.

3.3.2 Updating the Location Servers

The object servers need to keep the information at the authoritative location server up-to-date. For this we need a new protocol that is used to exchange information about new copies of objects. For example, when a cache caches a new object, it must send a message to the authoritative location server responsible for this object, indicating that the object has been cached. This message should also include the information which will be present in the resource record (e.g., last modification date) as well as a TTL-estimate. Similarly, when the cache removes an object, it must send a message to the location server indicating that the object is no longer cached.

Since the location data is managed over DNS, the dynamic update facility of DNS [87] can be used to dynamically update the location information. With this method, it is possible for a host to add or delete resource records atomically in an authoritative server. The host can specify update prerequisites if desired.

Using the dynamic update facility of DNS, an object server sends an update message every time the status of the object changes. For a Web cache this change of status could be the caching of a new object, a removal of an object or caching a new version of the object after the object has been modified at the origin server. In order to reduce the amount of update traffic, object servers can send their reports in batches. This is especially beneficial for Web caches that can send the information about an HTML page and all inlined images in one message, thereby reducing the overhead considerably.

3.3.3 Implementation

Implementing the LDS is relatively straight-forward and can be done incrementally. Since the system is an extension to DNS, no new servers need to be created and introduced.

For a DNS server to be LDS-capable, it needs to be able to interpret a new query type code and the associated resource records. DNS already handles numerous different query and resource record types, so adding one more is simple. Furthermore, the authoritative server has to be able to maintain the location information database and construct resource records from this database to include in replies to queries. Although this URL database contains more data than a normal DNS database, it can be maintained in a similar way. For a Web cache to be LDS-capable, it simply has to be able to send the update messages to the authoritative server.

The architecture allows for incremental deployment, since if a content provider does not operate a location server, no LDS resource records will exist. If no LDS resource records are available, the querying host would forward all requests using a static policy, for example, forwarding them directly to the origin server or a parent in a cache hierarchy. If LDS records exist, then LDS-enabled clients could use them to provide a better service for users. Non-LDS clients would again use the normal, static methods for finding objects. In the early phases of deployment, the traditional hierarchy should be kept as a fallback for clients and servers that do not implement the new protocols.

3.4 Network Web Caching

In this section we propose a new cooperative caching scheme that exploits the LDS. We assume that browsers forward their requests through a proxy cache in addition to the browser cache. This method is currently widely used by ISPs and other institutions, such as companies and universities, to offer a better service to their clients as well as to reduce the out-going traffic. Also, if a firewall is used, then all requests must go through a proxy at the firewall in any case, so adding a cache there does not present a significant overhead.

We now describe our cooperative caching scheme. A browser first sends an HTTP request for an object to its proxy cache. If the proxy cache does not have the object, the proxy cache invokes LDS to obtain a list of all the object servers (i.e., origin server and some other caches in the network) that contain the object. The proxy cache then chooses the "best" object server from the list and forwards the HTTP request to this object server (see Section 3.6). The proxy receives the object from the best object server and forwards the object to the browser.

Our caching scheme has several appealing features. First, at most two servers are visited to retrieve an object. Standard hierarchical caching schemes (such as NLANR [52] and Renater [68]) can cause requests to pass through a large number of object servers before a copy of the object is found, which can severely increase object retrieval time [85]. Second, the scheme allows the proxy server to choose from all the object servers that contain the object. Let us look at a couple of scenarios:

- LDS returns a list of object servers, with one cache server in a neighboring ISP of the ISP that contains the proxy cache, and all the other object servers on more distant ISPs. In this case, the proxy server would choose the cache in the neighboring ISP.

- LDS returns a list of object servers, with one cache server on the same continent as the proxy server, and all the other object servers on the other side of transoceanic links. In this case, the proxy server chooses the cache in its continent.

- LDS returns a list of object servers, with all the object servers far from the proxy and near or in the origin server's ISP. In this case, the proxy may still prefer to choose one of the caches over the origin server: The origin server may run on a slow machine or have a slow network connection.

It is important to note that traditional hierarchical caching does not permit the last option. With hierarchical caching, if there is a miss at the root cache, the root cache forwards the request directly to the origin server. Thus our scheme enables the proxy to select from *all* caches that are on the "route" between the proxy and the origin server.

Our caching scheme does have an unusual peculiarity. In the scheme just described, the proxy server will always retrieve an object either from the origin server or from some other proxy server. Because all the proxy servers are near the bottom of the Internet hierarchy (in institutional ISPs or local residential ISPs), with the exception of the origin server, all copies of an object are stored, essentially, in the leaves of the Internet. There are, however, several compelling reasons to also store copies of objects higher up in the Internet hierarchy, in particular in regional and national ISP caches. First, the requesting proxy may be able to retrieve an object more quickly if it is available from a common parent (or grandparent) of some other proxy cache that has the object. Second, hierarchical caching provides an asynchronous multicast infrastructure, which can significantly reduce bandwidth usage in the Internet [70].

Given that cache servers are present at regional and national levels, and peering agreements exist which organize the proxy, regional and national caches into hierarchies, we now modify our caching scheme to exploit the higher-level caches.

3.4.1 Populating Caches

In order to exploit the higher-level caches, we must make sure that the higher level caches obtain copies of the objects cached at lower levels. We propose that lower level caches push objects to their higher level parents. This way the objects are easily available for siblings under the same parent, and requests from distant hosts can be satisfied at a higher level in the network.

The details of our pushing strategy are as follows. A low level cache periodically sends a list of new objects (with last modification dates) to its parent. The parent decides which of the objects in the list to cache and retrieves these from the child cache. This is done at every level in the hierarchy and, in the end, the caches are filled up as they would have been using traditional hierarchical caching. Caches at the lowest level should send information about every object, but at higher levels a cache might use some locally defined policy (e.g., objects that are cached in multiple child caches) to decide which objects to report to its parent.

3.4.2 Network Traffic

Since LDS increases the number of messages sent over the network, it may overload some links or servers and in fact degrade performance. We are currently performing an analysis of how much LDS increases current network traffic. The following are the key factors:

1. **LDS queries and replies:** Although these are normal DNS messages, an LDS query is sent every time a proxy needs to retrieve a URL that it is doesn't have cached. Caching LDS records at the location servers will reduce some of this traffic; however, caching is not expected to have the same impact that it has with ordinary DNS, since a URL reference is more specific than a hostname reference.

2. **Update traffic:** When the status of an object changes in a cache, the cache must send an update message to the object's authoritative location server. For popular, widely cached objects, this may result in too much traffic directed at the authoritative server.

In order to reduce the amount of LDS lookup traffic in the Internet, we propose a variation to our basic LDS scheme. In this variation, once we get the location information for an HTML-page we assume that all the inlined objects on the page are also available from the same set of object servers. One advantage of this scheme is that it heavily cuts down on the amount of LDS traffic compared to that basic scheme since we now only send one LDS query for each HTML-page instead of each URL. The disadvantage of the variation is that there are no guarantees that the inlined objects are available from the same object servers. Origin servers and replicated servers should have the objects but a cache may have already purged some or all of them. If the object server does not have the requested object, we will do an LDS query for that object to obtain up-to-date location information.

In Section 4 we will show how the amount of LDS traffic can be significantly reduced when the LDS system is restricted to replicated servers.

To address the issue of update traffic, we propose that any cache that has a parent refrain from sending update information. Instead, the update information is forwarded to the parent during the pushing phase. The highest parent that does not have a copy of the object sends to the authoritative server an update message that contains update information for itself and its updated children. The benefit is that there are significantly less higher level caches than low level caches, thus much less update traffic.

3.4.3 Practical Considerations

In the above scheme, the highest parent in the hierarchy sends update messages for all its children which are thus all included in the database at the location server. As a result, institutional caches would also be included in the list returned by the location server, which means that they could receive requests for cached objects from hosts outside their own network. It is desirable to keep the institutional caches off the list of locations for two reasons. First, institutions likely do not want outside hosts using their bandwidth to retrieve objects. Second, objects at institutional caches are likely to be cached in the parent caches and outside hosts can retrieve objects faster from the parent cache.

Cache digests [73] are used in traditional hierarchies to represent cache contents in a compressed form. A cache fetches the digests of its neighbor caches and when a request cannot be satisfied locally, the cache checks the digests of the neighbor caches to see whether any of them have the object cached. If so, the object is retrieved from that cache; otherwise the request is forwarded to the parent cache. In LDS-based caching, having the digest of the parent cache is useful because this way we avoid the LDS-lookup for objects that are cached at the parent and would in any case be retrieved from there.

3.5 Replicated Servers

The LDS scheme is not limited to caching. It can also be used to disseminate information about replicated or mirrored objects. When an object is replicated at another server, this information is added into the location database at the authoritative location server. When a client requests this replicated object, it does an LDS lookup and gets a list of all servers holding a copy of the object. Because the information is not dynamically replicated as in caching, but rather placed at well-chosen locations, there is rarely need to notify the location servers of new copies. Likewise, there is no need to push objects into other servers since all replication decisions are made off-line.

With LDS, a user addresses a replicated object by the object's unique URL, LDS returns the list of servers that have the object. The browser then transparently chooses one of these (the best in some sense). The user would not have to know about the actual physical location of the object.

3.5.1 Reducing LDS Query Traffic

The scheme just described for replicated servers still requires that clients send an LDS lookup for each URL. In order to reduce the number of LDS messages we propose the following method of replicating servers and storing information about the replicated URLs. In this scheme, the LDS no longer keeps track of cached objects. It only tracks replicated and partially replicated servers. We require that replicated servers replicate either whole sites or complete sub-trees of servers. An example sub-tree of an origin server is all the objects on the origin server below a given URL prefix, e.g., all objects under `http://cnn.com/sports/`.

Suppose that a client wants to retrieve a URL. Normally, a client would send a DNS lookup to obtain the IP address of the origin server and retrieve the object from there. Using LDS the client proceeds as follows. First, the client sends an LDS query for the hostname in the URL. The LDS system returns a list of all servers that mirror either the whole site or a complete sub-tree from the site. We need to extend the resource record defined in Section 3.3.1 to include a prefix indicating the sub-tree that is mirrored by that particular server. The client then chooses the "best" server among those servers that mirror the desired sub-tree and forwards the actual HTTP-request to this server. A server may mirror multiple sub-trees from a single origin server; in this case LDS would return one resource record for each sub-tree.

Compared to the caching scheme discussed in Sections 2 and 3, this scheme has several advantages. First, and most important, in this mirroring scheme the client sends only one LDS lookup for each server; in the caching scheme, the client must send an LDS lookup for *each* URL. The caching scheme drastically increases DNS traffic in the network while the mirroring scheme keeps DNS traffic at its current level. (Recall that a client would in any case have to do a DNS lookup to get the IP address of the origin server.) Second, objects at mirrored servers tend to stay at the mirrored servers for long periods of time while objects in caches are cached and purged dynamically. Hence, LDS information for a mirrored server can be cached longer at location servers which greatly reduces lookup traffic. When the authoritative LDS server sends information about a mirrored site, it should first send out information about mirrors that mirror the most of the site. This is because DNS replies are by default sent over UDP and the message size is severely limited (512 bytes). By sending information about mirrors that mirror large subtrees, we maximize the likelihood that the client has the necessary information in the reply.

Compared to the standard DNS-way of accessing objects, our mirroring scheme does not increase the number of DNS messages in the network. The reply messages are slightly larger, however, since we need to indicate the prefix in the resource record. We do not expect this increase in size to be significant since URL prefixes can be efficiently encoded using the encoding specified in Section 3.3.1 and DNS resource record pointers.

3.6 Routing Decision

When the cache has received a reply to its LDS-lookup containing several object servers (caches and origin servers), the next question is: "Which one of the possibilities to choose?" Determining the best alternative is an important topic of our ongoing research.

Possible solutions include:

- All querying hosts measure connection qualities to object servers and keep a list of servers with known good performance.
- Pass QoS information along in LDS updates and replies.
- Use explicit routing information from BGP.

We will now provide some details on how these methods could be used.

In the first option, all querying hosts (e.g. low level Web caches) measure the time it takes to fetch objects from other servers (Web servers or other caches). This download time gives a crude estimate of the actual bandwidth between the two hosts. The querying host keeps a list of servers with which it has had good connections and prefers those servers to others when both types of servers are present in the LDS reply. Of course, the actual network conditions change all the time, but the results presented in [50] on performance characteristics of mirror servers indicate, that from a large set of mirror servers, only a small number need to be considered as candidates for download. Although the study in [50] uses a fixed number of mirror sites, we believe that the results can be applied to situations where the number of servers changes dynamically.

The second option is to pass some QoS information in the LDS replies along with the age information about the object. For example, a Web server can measure the connection times of incoming connections, estimate the connection bandwidth and take an average of the estimated bandwidths over all connections. This average bandwidth can be seen as the bandwidth that a random client somewhere in the network could expect to get when requesting objects from this server. Likewise, caches can measure the bandwidths of all out-going connections and calculate the average bandwidth.

The third option uses explicit routing information obtained over BGP by talking to local routers. This information gives the host a topological map of the network which can be used to find out how far each of the servers in the LDS reply are. Unfortunately, routing information gives only reachability, not quality, so some quality measures, as in the first two options, would be necessary. Grimm et al. [29] have studied using routing information in request routing and have decided against using BGP information because of configuration and security issues, amount of data and difficulty of obtaining it. Their approach is based on using whois-services.

Regardless of the approach chosen, any querying host can implement any local policies necessary when deciding where to forward a request. For example, a cache operator may want to forward requests to other caches based on the domain of the origin server.

3.7 Related Research

In [24] Gadde *et al.* . compare traditional hierarchical caching with an architecture where a single centralized server holds information about where each document is cached. When a low level cache wants to find out where an object is cached, it sends a message to this central mapping server which responds by either redirecting the request to a peer server or by forwarding it to the origin server. This scheme results in all of the objects being cached at institutional caches. The authors also perform simulations using a small number of caches and find out that the performance of the centralized solution is on average better than that of the traditional hierarchy. The number of caches in the simulations is low, only 8 and 32 cache configurations are simulated. The authors express their doubts about the scalability of their solution, but present some ideas for replicating and distributing the location information.

In our approach, the querying host gets a list of *all servers* holding a copy of the object instead of being redirected to one of them by a mapping server. This puts the querying host in control over where the request will be forwarded. This decision might greatly depend on local conditions and policies unknown to a central server. Another difference in our approach is that the location information is distributed over DNS which is ubiquitous and provides satisfactory service.

Tewari *et al.* [85] present an architecture where data is cached near the browser and location hints are passed through a metadata hierarchy. Their architecture arranges the caches in virtual hierarchies, one for each object, for distributing the metadata information. In their architecture, when a cache caches a copy of an object, it sends out a location hint indicating the URL and its address. This hint is propagated in the virtual metadata hierarchy and could eventually reach all caches in the system. Caches keep track of all the hints they have received and use them to forward requests. If a cache receives a request for an object which is not cached locally but the hints indicate another cache holding the object, the request is forwarded to this cache. The virtual metadata hierarchies are constructed using IP-addresses and URLs in a way which tries to minimize the distances between parents and children. Also, the hierarchy guarantees that a cache can receive only one hint for any object. This hint is likely to be from the nearest cache holding the object, but this is not guaranteed.

One difference between their and our approaches is the way caches are placed. In their architecture only institutional caches hold objects and every request forwarded using a hint would be forwarded to another institutional cache. This behavior might be unacceptable to some people who do not want more external traffic on their networks. In our approach, objects are pushed to higher levels and this eliminates the need for going to other institutional caches. Second important difference is that using LDS for locating objects, a querying host gets a list of all possible locations. It can then choose from the list according to a local policy or by adapting to current network conditions. In [85] a cache receives only one location hint. Since the virtual metadata trees are based on IP-addresses and URLs, there are no guarantees that this hint indicates a good server. We are currently performing a comparison of the two schemes.

Amir *et al.* [5] study three different methods for redirecting requests to mirrored sites – HTTP redirection, DNS round trip time measurements, and shared IP addresses. In their DNS-based solution they use DNS round trip time measurements to determine the best replicated server for a client. They place the replicated servers near an authoritative DNS server which gives the address of the replicated server when queried for the origin server. This results in clients being eventually redirected to their closest replicated server. There are two major differences between their approach and our replicated server approach (Section 3.5.1). First, in their system replicated servers must always replicate the entire site while in our system a server may replicate only sub-trees of the original site. Second, in their system the replicated servers must be placed in close proximity of the authoritative DNS server. Our architecture places no constraints on the placement of the servers.

3.8 Future Work

We will continue our work on LDS by performing quantitative comparisons of the different architectures presented in this chapter. In particular we will concentrate on evaluating the amount of new traffic in the network caused by the lookup and update messages. We will also compare the traditional caching hierarchy with our different LDS architectures to find out how much object access latency can be reduced through LDS. We will also closely study the request routing issue in order to find out how much information is needed to make good routing decisions.

3.9 Conclusion

In this chapter we have presented the Location Data System, a new network application that can be used to locate where copies of objects are stored on the Web. LDS provides a service similar to DNS and can be implemented as an extension to DNS and deployed incrementally. We have presented two applications of LDS, locating objects in mirrored servers and locating objects in Web caches. We have also discussed how clients can decide the best server to forward requests to based on information on network conditions and topology collected from the dynamically from the network.

Chapter 4

Replicated Directory Service

4.1 Overview

We propose a new design for the Domain Name System (DNS) that takes advantage of recent advances in disk storage and multicast distribution technology. In essence, our design consists of geographically distributed servers, called replicated servers, each of which having a complete and up-to-date copy of the entire DNS database. To keep the replicated servers up-to-date, they distribute new resource records over a satellite channel or over terrestrial multicast. The design allows Web sites to dynamically wander and replicate themselves without having to change their URLs. The design can also significantly improve the Web surfing experience since it significantly reduces the DNS lookup delay.

4.2 Introduction

We propose a new design for DNS that takes advantage of recent advances in disk storage and multicast distribution technology. This design can be implemented incrementally, allowing for a graceful evolution from the current DNS to a new system. Our design does not change the syntax or semantics of the DNS messages; it only affects the way the DNS database is managed. Our proposed design has two principal features:

- First and foremost, it is highly responsive to changes in DNS information. Being able to rapidly change DNS information and propagate the changes is useful in solving the hot-spot problem on the Web. Often, a single web server becomes suddenly popular and receives many more requests than it can handle. In our design, without changing URLs, the web server could replicate its contents on another web server and inform the DNS system of the alternative server. Because the new web server is immediately available in the DNS to every client, many clients would likely choose it over the old server (by using, for example, DNS rotation), thus significantly reducing the load on the server and improving the quality of service for everyone. Modern DNS does not allow this because nameservers are allowed to cache resource records.

- It can significantly reduce the DNS lookup time. The system eliminates the need to query distant authoritative servers. Because DNS round-trip time is often a significant fraction of the delay when accessing a Web page, our design can improve the Web-surfing experience.

Our design is inspired by two recent technological advances. First, disk storage has become abundant and cheap in recent years, and the trend is expected to continue. As we shall argue in the body of this chapter, the entire DNS database can be stored in disk of a household PC. Second, emerging terrestrial multicast and satellite broadcast systems can efficiently distribute DNS information. We note here that these two advances abundant disk space and efficient broadcast distribution are currently being exploited by Web cache technology in order to bring the "Web to the edge of the network" [80].

In essence, our design consists of geographically distributed servers, called replicated servers, each of which having a complete an up-to-date copy of the entire DNS database. To keep the replicated servers up-to-date, they distribute new resource records over the satellite channel (or over terrestrial multicast).

In this chapter we provide (*i*) a detailed description of the design, (*ii*) a detailed feasibility analysis for network traffic, replicated server traffic and disk storage requirements, and (*iii*) a plan for migrating the existing DNS system to our replicated design. We also discuss the security and fault-tolerance issues for this new design. This chapter is organized as follows. In Section 4.3 we provide a short overview of the existing DNS system. In Section 4.4 we describe the replicated DNS architecture. In this section we also analyze storage and network bandwidth requirements for the new architecture. In Section 4.5 we model the tradeoff between the staleness of the DNS information and the traffic load at the replicated servers. In Section 4.6 we use empirical Internet data to study how much our architecture will reduce the time it takes to resolve a hostname. In Section 4.7 we look at how the existing DNS system can migrate to our proposed architecture in a graceful manner. In Section 4.8 we address security and fault tolerance issues for the new architecture. Finally, Section 4.9 concludes the chapter.

4.3 Overview of DNS

In this section we provide a brief overview of DNS and introduce some terminology that we shall use throughout the chapter. The principle task of the DNS is to provide a mapping from the human readable domain names to numerical IP-addresses used to identify hosts on the Internet [45, 46]. It is implemented in a distributed database consisting of a hierarchy of nameservers. The name space is divided into zones and each zone has two or more *authoritative nameservers* that are responsible for keeping information about that zone up-to-date. One of these authoritative servers is the *primary nameserver*, which holds the master file containing all the resource records for that zone. When new hosts are added to a zone, the administrator must edit this file manually to make the new hosts public. The other authoritative servers (*secondary nameservers*) periodically fetch the contents of the master file in order to keep their records up-to-date. These zone transfers are done using the special zone transfer query type in DNS (AXFR query type [46]).

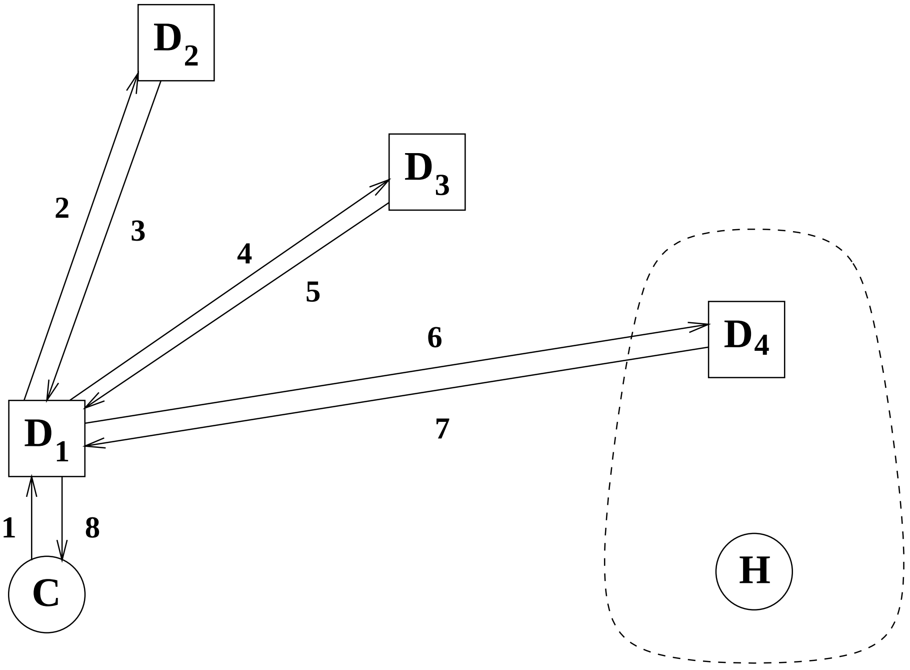


Figure 4.1: A DNS query. C is the client, H is the host that the client is trying to resolve, D_1 is the local nameserver, D_2 is a root name server, D_3 is an intermediate nameserver, and D_4 is an authoritative nameserver for the queried host H.

When a client needs to obtain an IP-address for a hostname, the query proceeds as shown in Figure 4.1. In Figure 4.1, C is the client, H is the host that the client is trying to resolve, and D_1–D_4 are DNS servers.

First, the client sends the query to its *local nameserver* D_1. Typically this local nameserver acts as the primary nameserver for the zone where the client resides and has all the DNS information for that zone as well as cached copies of DNS information from other zones that the local clients have recently queried. Assuming that this server does not have a cached copy of the information, it queries one of the *root nameservers*, D_2 (currently there are 13 root nameservers in the world [71]) which returns a referral to a nameserver responsible for the top-level domain of the hostname. The local nameserver then queries this server D_3 and gets a referral to an authoritative server D_4 for the domain in which the host is located. Finally, the local nameserver queries the authoritative nameserver and gets the reply with the IP-address of the host. When the local nameserver receives the reply it sends it to the client and caches a copy. If another client now wants the address of the same host, the local nameserver can immediately return the cached copy, thus avoiding the need to query distant nameservers. (In this example we assumed that all the queries from D_1 are iterative; recursive and combinations of recursive and iterative queries are possible. We also assumed that there is one intermediate nameserver between the root and the authoritative nameserver; in reality there can be more or less.)

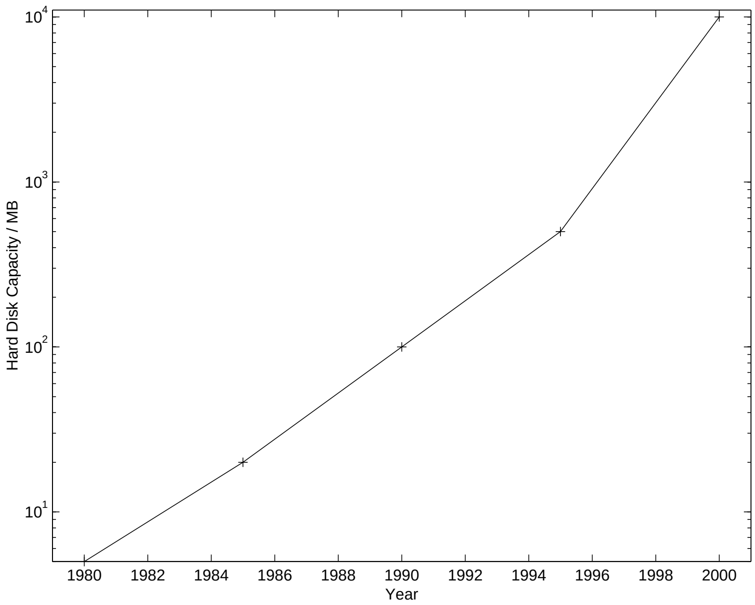


Figure 4.2: Evolution of Hard Disk Storage Capacity

4.4 Replicated DNS Architecture

Our architecture makes use of *replicated nameservers*, with each replicated nameserver storing the *entire* DNS database. Before describing the architecture in detail, we perform a simple feasibility analysis for storing the entire DNS database on a nameserver.

4.4.1 Storing the Entire Database on a Server

Server disk storage has experienced dramatic increases over the past twenty years. Figure 4.2 shows the evolution of disk capacity on a typical desktop computer from 1980 to 2000.

In 1999 standard PCs are sold with approximately 10 gigabytes of disk storage. If storage capacity continues to grow at current rates, standard PCs will be sold with 20-30 gigabytes of storage in 2000.

Now let us estimate the storage requirements for a replicated nameserver. Since each replicated nameserver replicates all of the DNS information on the Internet, it must have an entry for every host in the DNS. It is possible that a hostname maps to multiple IP-addresses, but this is not very common, so, for simplicity, our analysis assumes that we need one resource record for each existing hostname.

The Internet Domain Survey [31] reports that in July 1999 there were slightly over 56 million hostnames registered in DNS. Using the data from their January 1997 survey we can calculate the average length of a hostname in the DNS which is 20.07 characters. We will conservatively assume that we need 40 bytes of storage per hostname; this includes the actual hostname, IP-address, TTL-information, and possibly other information. In

1 gigabyte of storage we can store entries for 25 million hostnames, thus we would need a bit over 2 GB of storage to store the IP-addresses of all the hostnames in the DNS in 1999.

The above estimate only accounts for IP-addresses and hostnames, i.e., only for DNS A-type resource records. Note that NS-type resource records are not necessary, since the replicated nameservers contain information about all hosts. We will, however, need to account for other types of resource records, most notably PTR, CNAME, MX, and SOA records. In order to get an upper bound estimate, we will assume that there is one resource record of each of these types per host. In reality, only PTR-records exists for each host. Most hosts on the Internet do not have CNAME records, MX records exist usually on a per domain basis, and SOA records exist only for authoritative nameservers. According to our conservative assumptions, we would need 10 GB of storage to store all the information in the DNS system. In reality, this would probably be much less, on the order of a few gigabytes, which is easily stored on the disk of an inexpensive PC.

4.4.2 Interaction between Authoritative and Replicated Nameservers

Our proposed architecture consists of the current authoritative primary nameservers and a number of replicated nameservers (something between 10 and 10,000) distributed all over the world. Ideally there should be at least one replicated nameserver per region (e.g., one per country) so that clients can receive replies to queries fast. Replicated nameservers could also be present at the local ISPs and at corporate and university networks.

Our replicated architecture maintains the “local administration, global availability” philosophy behind modern DNS [4]. In our replicated architecture the primary nameservers still keep track of their own zones in the normal way, i.e., they have the master file for the zone and administrators make changes to this file in the usual manner. However, our architecture replaces the secondary authoritative servers with replicated nameservers. A replicated nameserver is responsible for all the primary nameservers in its region and periodically fetches the zone information from the primary nameservers. Naturally, a replicated nameserver can also replace a primary nameserver, if needed. (For clarity of presentation, we assume in the following that each primary nameserver is associated with a single replicated nameserver; in reality the primary nameserver could interact with any number of replicated nameservers.) In the new architecture, the primary nameserver of a zone also pushes the information to its closest replicated server when the information in a zone changes. This combination of pushing updates and periodically fetching the whole zone guarantees fast updates and guarantees that the information is refreshed periodically in case some of the update messages were lost in the network. (Recall that by default DNS works over UDP which offers no guarantees.) It also provides a mechanism for replicated servers to quickly recover from a crash. For the fetching, the replicated server uses the normal DNS zone transfer query (AXFR query type; zone transfers are done over TCP); for the pushing the primary servers could use the DNS dynamic update method [87].

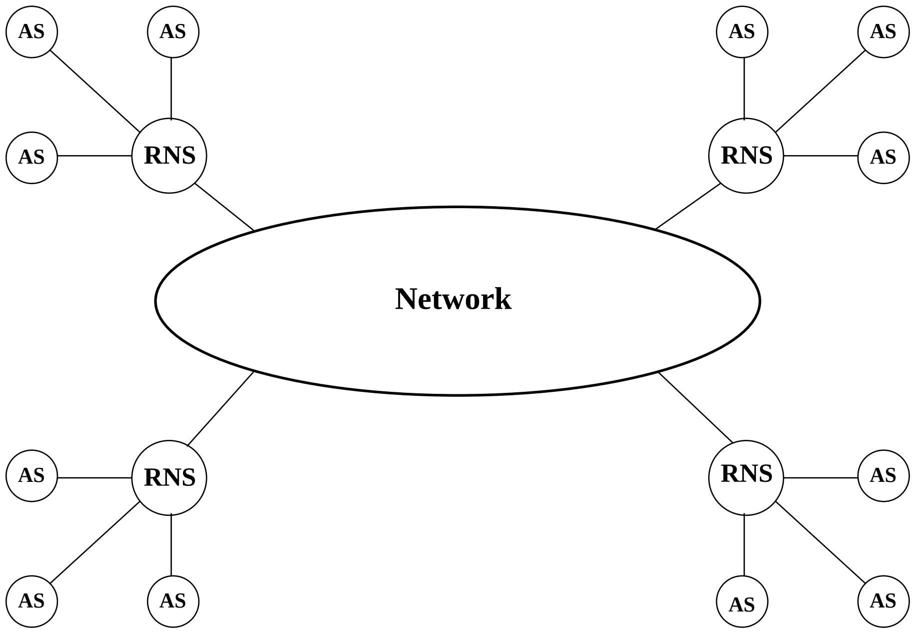


Figure 4.3: Connections between authoritative nameservers (AS) and replicated nameservers (RNS)

4.4.3 Interaction among Replicated Nameservers

The replicated nameservers contain an up-to-date database of the entire DNS database. As shown in Figure 4.3, this is accomplished by having the replicated nameservers communicate with each other in order to share the updates they have received from the primary nameservers that they parent. In this section, we outline two schemes for distributing the update information: multicast and satellite.

IP Multicast

In this scheme the replicated servers all belong to a multicast group which has a single multicast address. When a replicated server receives new information from its child primary nameservers, it sends the information into the multicast address. Eventually all replicated servers will receive the update. The advantage of this solution is that it does not require any special hardware, but it needs all the replicated nameservers to have access to multicast, e.g., to be connected to MBone [19]. This solution works best if the number of replicated servers is not too large and they can be equipped with multicast; also the amount of update traffic may become an issue since the update traffic has to share the bandwidth to the replicated nameserver with the normal query traffic.

Satellites

Another possibility for transmitting information between the replicated servers is to bypass the Internet and send the information over a satellite channel. Because of the broadcast nature of satellite and our need to broadcast the information to all replicated servers, a satellite channel is a very attractive alternative for distributing the information.

The main advantages of using a satellite channel for distributing the information are the following: (1) The update traffic does not have to travel long distances over the terrestrial Internet; (2) information is immediately available at all replicated nameservers.

We propose two possible schemes for handling the use of the satellite channel. In the first scheme one of the replicated nameservers is designated as the central server. All the other replicated nameservers collect the updates from the primary nameservers in their own regions, aggregate the updates, and send them over the Internet via unicast to this central server. The central server receives updates from all other replicated servers and broadcasts these updates over a satellite link. This scheme requires that all replicated nameservers be equipped with a satellite receiver. This method has the advantage that there will be no collisions on the satellite channel since only one station is transmitting. This architecture is similar to the SkyCache architecture [80] where Web caches send access reports to a central master server which streams the most popular Web pages over the satellite link.

In the second scheme we have no central replicated server. Instead each replicated nameserver has a satellite transmitter and broadcasts over the satellite channel the updates it has received to all other replicated nameservers. In this scheme we have to use some method of resolving the conflicts that might occur when two different servers want to broadcast information at the same time. According to [60] the best medium access control protocol for bursty traffic is either random access or a simple reservation based protocol, depending whether the messages are short or long, respectively. We expect DNS update traffic to be bursty and the messages to be relatively short, so a simple medium access protocol, such as Aloha or Slotted Aloha, is likely to provide sufficiently good performance.

The choice of the update scheme depends greatly on which kinds of satellites, GEO or LEO, we are using. In order to get global coverage with geostationary satellites, we need to use several satellites in a coordinated fashion. Therefore it is simpler to use the second scheme with GEO satellites. In a LEO constellation, the centralized scheme works as described above since the constellation provides global coverage; the second scheme in a LEO constellation would most likely be analogous to terrestrial IP multicast.

One downside of using satellites, compared with using IP multicast, is that we need to install satellite receivers and transmitters and obtain bandwidth on the satellite channel. The first scheme only requires a satellite receiver on the replicated nameservers; this can be a normal satellite dish, such as the ones used in the DirecPC-service [17]. Such dishes are relatively cheap, around \$300–\$400, and provide several hundreds of kilobits/second of bandwidth (DirecPC operates at 400 kbps/s). The second scheme is more expensive since we need to install both receivers and transmitters; typically this is done with Very Small Aperture Terminals, or VSATs. An average VSAT costs between \$5,000 and \$10,000 making the cost of a single replicated nameserver much higher than in the first scheme. Note, however, that in the first scheme we need also the central up-link station which increases the total cost of the installation. These costs are estimates for a system using geostationary satellites.

Reliability

Because having correct information in the replicated nameservers is vital to the DNS system, we must ensure a reliable means of distributing the update messages. This means

Bandwidth (kbit/s)	Centralized	Aloha	Slotted Aloha
128	4.83	0.87	1.74
256	9.68	1.74	3.48
512	19.35	3.48	6.97
1024	38.71	6.97	13.93
2048	77.41	13.93	27.87

Table 4.1: Number of updates per host per week for different satellite links

that some sort or reliable multicast protocol must be employed, whether terrestrial or satellite multicast is used. Protocols using Forward Error Correction (FEC) along with a return channel (which is clearly available for replicated DNS) can efficiently provide reliability [34, 58].

4.4.4 Traffic Analysis for Communication among Replicated Servers

Assuming that each resource record is 40 bytes and that the central master replicated server is using a 128 kbit/s satellite channel to broadcast information to other replicated servers, we can send information about 400 modifications every second. Assuming that these modifications are spread uniformly over all hosts, the information about a host could change every 110,000 seconds on the average, or once per 1.3 days. This is less than the default time-to-live period of 2 days in the popular BIND-nameserver. We can therefore safely assume that the rate of change of DNS information is far less than what can be handled over a 128 kbit/s satellite link.

Table 4.1 shows the number of updates per host on the average for different satellite link bandwidths. We show the numbers for the centralized architecture and the distributed architecture using both pure Aloha and slotted Aloha. Recall that the throughput for Aloha is 18 % and for slotted Aloha it is 36 % [60]. In [34] it is reported that using the FEC-protocol from [58] we can expect to use about 10 % extra bandwidth due to FEC.

The numbers in Table 4.1 assume that the changes are evenly divided over all hosts. In reality this is unlikely, since most hosts do not change their DNS information at all. The hosts that rapidly change their associated DNS information may require high rates of change. Assuming that 90 % of the hosts on the Internet are “stable”, i.e., they never change their DNS information, the remaining 10 % of the hosts could change their DNS information 48 times per week using the 128 kbit/s link; this corresponds to roughly 7 changes per day, or one change every 3.5 hours. Using a 1 Mbit/s link, the rapidly changing hosts could change DNS information 55 times per day, or little over twice an hour on the average.

4.4.5 Resolving DNS Queries

We discuss two variations for resolving DNS queries. Neither variation requires any changes in client DNS software. In the first variation, a client directly queries its replicated nameserver. (Thus, the client configures the client DNS software to point to its parent replicated nameserver rather than to a local nameserver.) Because the replicated nameserver contains a complete and and up-to-date copy of the entire DNS database, the replicated nameserver will be able to directly return the requested RRs. Furthermore, because the replicated nameservers are "close" to the client, the responses should come back fast.

One problem with the variation just described is that request load on a replicated server can be high if the load is not balanced over a large number of replicated servers. Our second variation for resolving DNS queries makes use of the existing infrastructure of local nameservers. In this variation, a client resolves a hostname as follows. First, the client sends a DNS query to its local nameserver. If the local nameserver is also the primary nameserver for the local zone and the query is for a local hostname, the local nameserver replies with the requested information. If the query is for a remote hostname and the local nameserver does not have the information cached, it sends the query to the nearest replicated nameserver. This phase is similar to a nameserver sending a query to a root nameserver with the exception that in our architecture the closest replicated server is typically well-defined; in normal DNS the nameservers measure round-trip times to other nameservers and make decisions based on past experience. (If the local nameserver has several replicated nameservers to choose from, it can use round-trip time measurements to select the closest one.) The replicated nameserver replies with the requested information, which the local nameserver caches and returns to the client. This scheme has the advantage that if the information is not cached, the query is sent to a replicated nameserver *in the same region as the client* instead of being sent possibly to the other side of the world as can happen in normal DNS. This greatly reduces the latency that a client observes when performing DNS queries.

In this second variation, a replicated nameserver receives less request traffic because many client requests are filtered by the local nameserver. In particular, all queries for a local host (i.e., in the same zone as the requesting host) and all queries for cached RRs are filtered. The downside of the scheme is that RRs in the nameserver caches can be stale (as is the case in the current DNS). Because one of our principle goals is to provide a DNS architecture that is highly responsive to hostname changes, the issue of staleness is important. In Section 4.5 we will study in detail the tradeoff between the amount of traffic at a replicated server and the probability of receiving a stale RR.

4.4.6 Arpanet Name Resolution

This architecture is a step towards the old Arpanet `HOSTS.TXT` -solution [4] with some important differences. In Arpanet, the `HOSTS.TXT` file, which was maintained on a single computer at Stanford Research Institute (SRI), contained the name-to-address mappings for all hosts on Arpanet. Administrators e-mailed changes to SRI and periodically transferred the `HOSTS.TXT` file over FTP. As Arpanet grew, several problems with this centralized solution emerged. First, the traffic load on the computer holding the master file became unbearable. Second, maintaining the consistency of the file across the network was difficult. Third, as more hosts were added to the Arpanet, the `HOSTS.TXT` occupied more and more storage in the server. In summary, the centralized mechanism didn't scale. Instead, a new system was designed to allow local administration of the data yet make the data globally available; this was the modern DNS.

Our replicated architecture takes the old HOSTS.TXT-approach by collecting rather than partitioning the entire DNS database. However, because of its replicated structure, our architecture scales well. When the modern DNS was designed in the early '80s disk storage was expensive; therefore maintaining a replicated database of all hosts on the network was infeasible. Nowadays, disk storage is cheap, and as we previously argued, it is quite feasible to construct a replicated database for all of the information in DNS.

4.5 Staleness vs. Traffic at Replicated Nameservers

In this section we suppose that clients direct their DNS queries to local nameservers, and a query is only forwarded to a replicated nameserver when there is a "miss" at the local nameserver (i.e., the second variation as described in Section 4.4.5). Passing queries through local name servers can significantly reduce the query traffic at the replicated servers. However, because the local nameserver caches RRs, there is a risk that the local nameserver will frequently reply with stale RRs.

When a Web site with a particular hostname is moved or copied to a new location, the RRs associated with that hostname change. If the Web site knows its locations will change in exactly t' seconds, then the TTL for the RRs can be set to the remaining lifetime of the RRs. This ensures that local nameservers never deliver to clients stale RRs for the site. However, in order to respond to randomly occurring hotspots, many sites will want to make spontaneous changes to their resource records. If the TTL for the RRs of such a site is set to a large value there is a risk that nameservers caching the RRs will frequently respond with stale RRs. On the other hand, if the TTL is set to a small value, then a large fraction of the queries will be forwarded to the replicated server. In this section we quantify this tradeoff.

For simplicity, we consider only the clients under a single local nameserver accessing one resource record (RR). This scenario is depicted in Figure 4.4. The local nameserver either has a cached copy of the resource record, or has to query a replicated nameserver for an up-to-date copy of the resource record. When the local nameserver queries the replicated nameserver, the replicated nameserver indicates a time-to-live value in the reply. We consider the following two criteria:

1. The fraction of queries from the clients to the local nameserver that receive a stale RR because the RR has changed.

2. The number of extra queries to the replicated server which only serve to validate the cached copy at the local nameserver.

Denote the rate at which clients request the resource record as λ. When there is a large number of independent and active clients under the nameserver, we can reasonably assume that the inter-request times are exponentially distributed. Denote by T the amount of

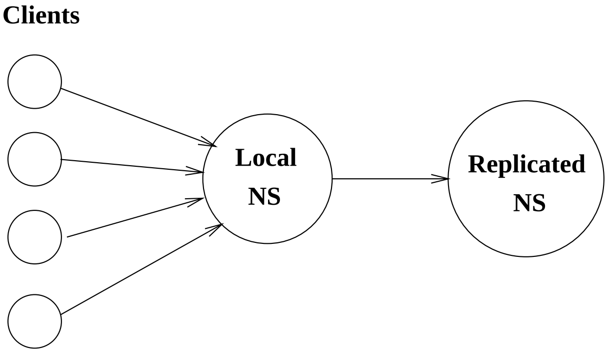


Figure 4.4: Clients, Local Nameserver (NS), and Replicated NS

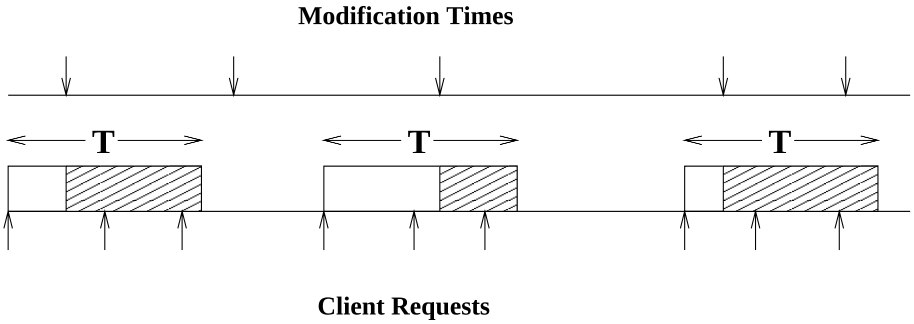


Figure 4.5: Client Requests and Modification Times

time for which the local nameserver is allowed to cache the resource record. Finally, let μ denote the rate at which information in the RR changes. We assume that the times between modifications are exponentially distributed.

Figure 4.5 shows the client queries and modifications to the resource record. The modification times are at the top and client queries are at the bottom. The boxes, each of length T, indicate the period during which the RR is cached at the local nameserver. The shaded boxes indicate the periods during which the clients receive a stale RR from the cache because the RR has changed.

We now calculate what fraction of requests are forwarded to the replicated nameserver. Given the above assumptions, the expected time between successive requests to the replicated nameserver is $T + \frac{1}{\lambda}$. Therefore the rate of requests to the replicated server is $\frac{1}{T+\frac{1}{\lambda}}$. Given that the total rate of requests is λ, the fraction of requests forwarded to the replicated nameserver is

$$\frac{1}{T+\frac{1}{\lambda}}/\lambda = \frac{1}{\lambda T + 1}. \tag{4.1}$$

Figure 4.6 shows the fraction of queries forwarded to the replicated nameserver for different λ and T. In Figure 4.6(a) we show faster request rates and TTLs up to one hour and in Figure 4.6(b) we show the slower request rates and TTLs up to 6 days.

From Figure 4.6(a) we can see that if the request rate is high enough, on the order of one request every 10 seconds ($\lambda = 10^{-1}$), even a short TTL-value, such as 5 minutes, is sufficient to satisfy most queries from the cache at the local nameserver. Figure 4.6(b) shows that if the request rate is very low, the TTL-value has to be extremely high to provide any measurable reduction in query traffic to the replicated nameserver. The situation shown in Figure 4.6(a) represents a scenario where the resource record is very popular and the product λT is large, i.e., the expected number of requests in a TTL-period is high. Figure 4.6(b) corresponds to the case where the resource record is not very popular.

We now calculate the fraction of requests that receive stale RRs. Let Y denote the time between successive modifications. Within each interval of expected length $T + \frac{1}{\lambda}$, $E[N_{(T-Y)^+}]$ requests see stale resource records, where N_t is a Poisson process with rate λ. Thus the fraction of stale resource records within an interval is

$$\frac{E[N_{(T-Y)^+}]}{1+\lambda T} = \frac{\lambda E[(T-Y)^+]}{1+\lambda T}. \tag{4.2}$$

We can explicitly calculate the expectation:

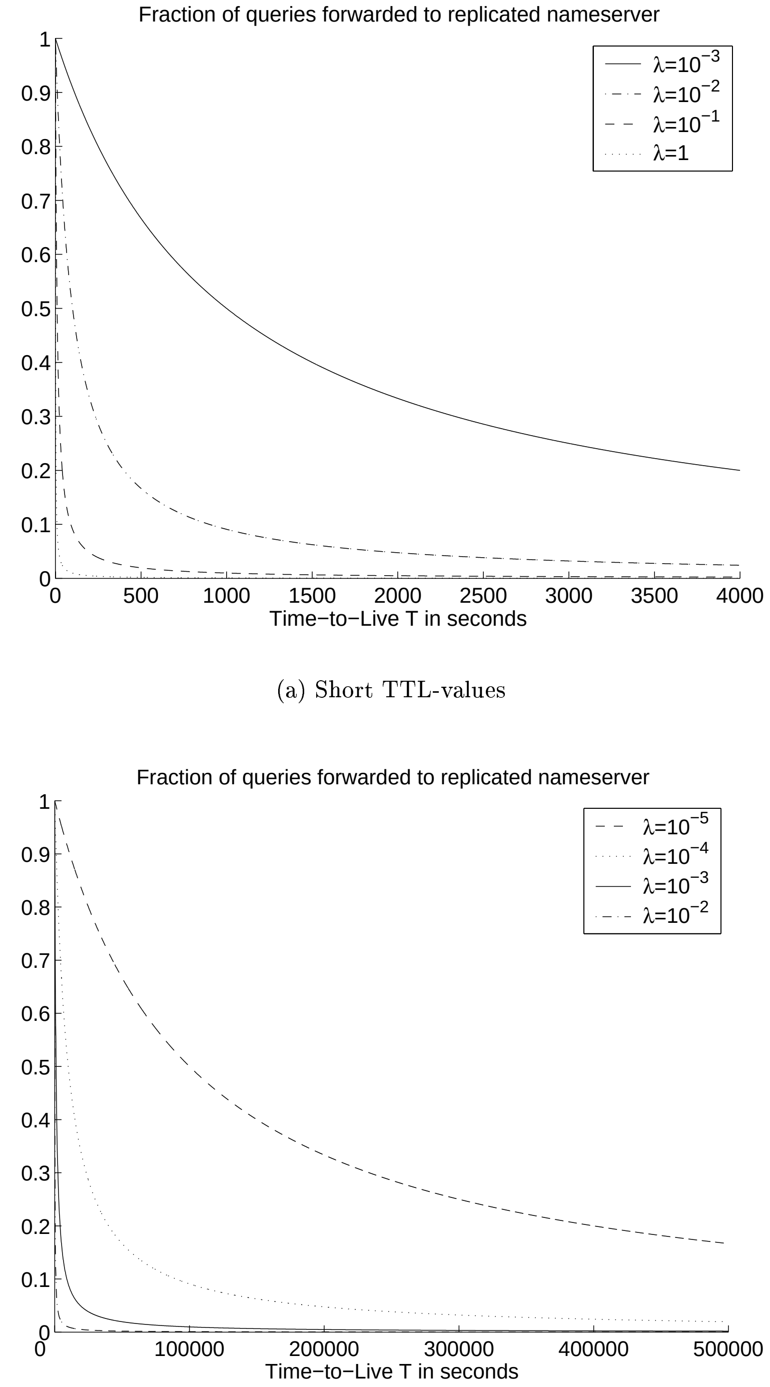
$$\begin{aligned} E[(T-Y)^+] &= \int_0^T (T-y)\mu e^{-\mu y} dy \\ &= \frac{1}{\mu}(e^{-\mu T} + \mu T - 1) \end{aligned} \tag{4.3}$$

The fraction of stale resource records is then

$$\frac{\lambda}{1+\lambda T}\left(\frac{e^{-\mu T} + \mu T - 1}{\mu}\right) = \frac{1}{\frac{1}{\lambda} + T}\left(\frac{e^{-\mu T} + \mu T - 1}{\mu}\right). \tag{4.4}$$

If $T >> \frac{1}{\lambda}$, i.e., we have a large number of requests in one TTL-period (λT is very large), then λ has very little effect on the overall fraction of stale resource records. In Figures 4.7 and 4.8 we show the fraction of stale RRs delivered to the clients in four different scenarios. In Figure 4.7(a) and Figure 4.7(b) we show rapidly and slowly changing RR, respectively and Figure 4.8(a) and Figure 4.8(b) compare the effects of λ on slowly changing RRs.

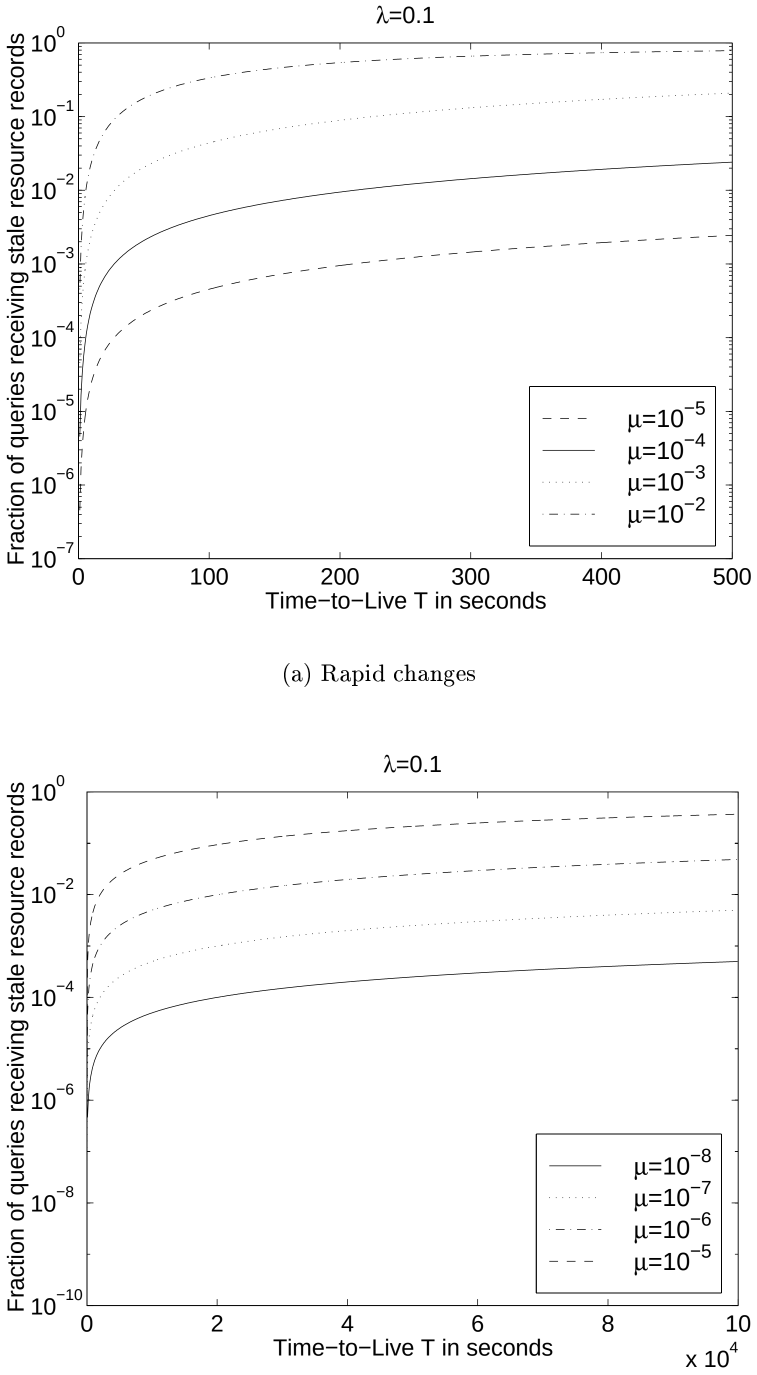
Assuming that we would like to have a fraction of less than 10^{-3} stale RRs, we can see from Figure 4.7(a) that this requires us to effectively disable caching at the local nameserver for RRs that change on the average more often than once every 3 hours ($\mu = 10^{-4}$). When the resource records are changing more slowly, as is the case in Figure 4.7(b), we can see that even for RRs changing on the average every 115 days ($\mu = 10^{-7}$), the maximum TTL, in order to have the fraction of stale RRs below 10^{-3}, is around 12 hours. Figures4.8(a) and 4.8(b) show that the more popular the RR is, the larger the fraction of stale RRs is (corresponding curves in Figure 4.8(a) are higher than in 4.8(b)). With longer TTL-values, the fraction of stale RRs becomes insensitive to λ, because the increase in T increases the product λT.



(a) Short TTL-values

(b) Long TTL-values

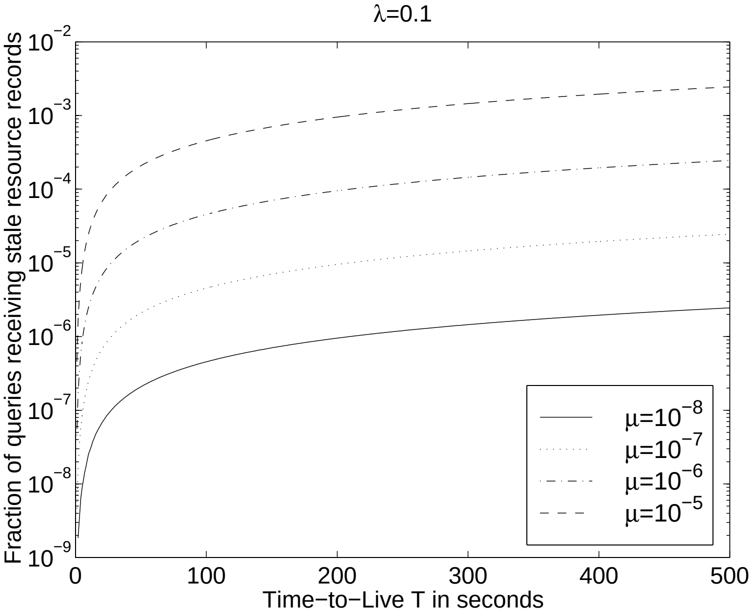
Figure 4.6: Fraction of queries forwarded to replicated nameserver



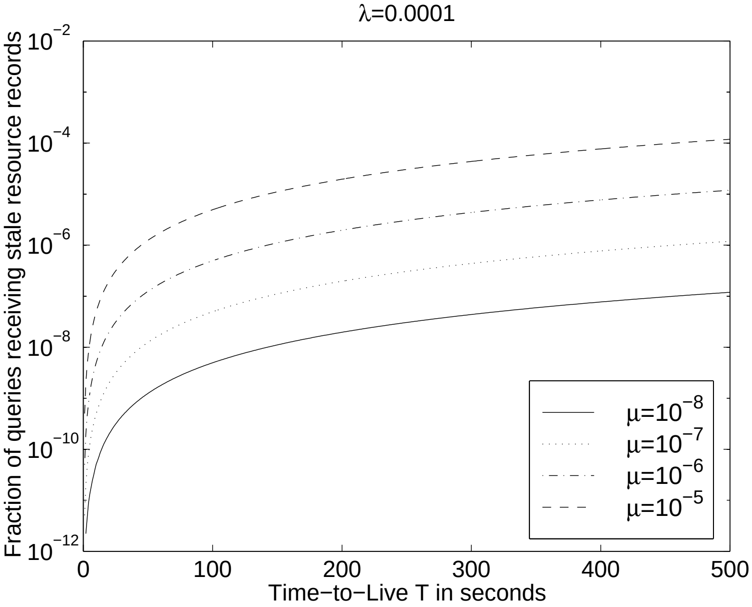
(a) Rapid changes

(b) Slow changes over long TTL

Figure 4.7: Fraction of stale resource records delivered to clients



(a) Slow changes, popular RR



(b) Slow changes, unpopular RR

Figure 4.8: Fraction of stale resource records delivered to clients (continued)

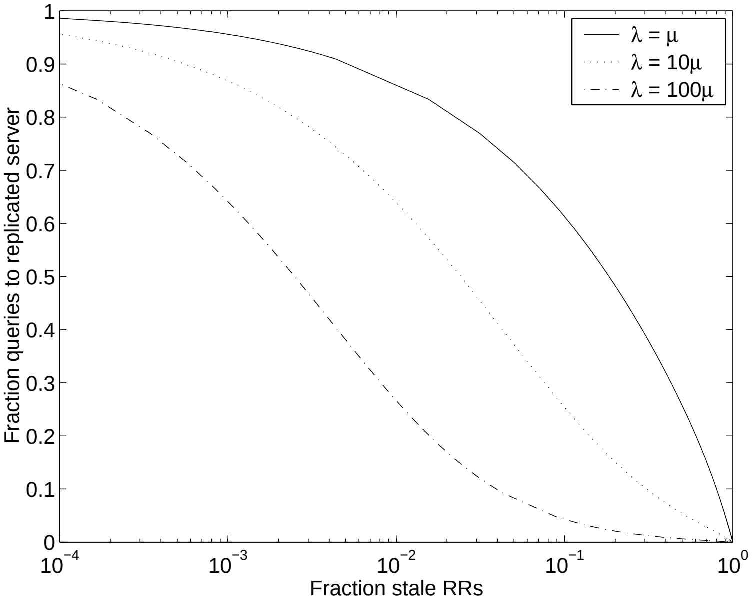


Figure 4.9: Comparing fraction stale RRs and fraction queries to replicated server

In Figure 4.9 we compare the fraction of stale RRs and fraction of queries to the replicated nameserver. The fraction of stale RRs is on the x-axis and the fraction of queries to the replicated nameserver is on the y-axis. We fix μ and λ ($\mu = 0.001$ and $\lambda \in \{0.001, 0.01, 0.1\}$; different values for μ and λ give similar results) and vary T to obtain the curves. As we can see from Figure 4.9, to get the fraction of stale RRs below 10^{-3}, we will have to forward around 65 % of the queries to the replicated nameserver, even in the case where the RR is requested frequently compared to its rate of change.

From these results we can conclude that replicated nameserver should allow the local nameservers to cache RRs for only short periods of time, on the order of a few minutes. This is because longer TTL-values increase the fraction of stale RRs considerably, and even a short TTL-value provides a sufficient reduction in query traffic to the replicated nameserver for popular RRs.

4.6 Latency Improvement

In this section we study how much our proposed architecture will reduce the time it takes to resolve a hostname. Because resolving some hostnames over the existing DNS requires contacting distant servers, a DNS lookup may introduce a significant delay into Web surfing and other network applications. To evaluate the delay in the existing DNS, we performed DNS lookups using the `host`- and `dnsquery`-commands on several different hostnames all over the Internet. In order to evaluate the effects of DNS lookups in a typical Web surfing context, we divided the hosts into two groups: popular and unpopular hosts. The popular hosts were the 35 most popular Web servers on the Hot100-list [30]. For the unpopular hosts

Site	**Average latency**	**Maximum latency**	**% of lookups exceeding 4 seconds**
FR	1.79	47.44	14.4
FI	1.85	42.2	15.5
US	2.10	75.4	6.6
All	1.90	75.4	12.5

Table 4.2: DNS query results for unpopular servers

we chose 200 Web servers randomly from 33 different top-level domains from the results of the Netcraft Web Server Survey [53]. We chose to study these two groups separately since the popular hostnames are likely to be found in the local nameserver and therefore the actual DNS architecture in the background has no effect. The less popular hostnames require the local nameserver to go out on the network to find the requested information. We performed the queries from three sites, one in France (FR), one in Finland (FI), and one in the US west coast (US). We discarded queries which either resulted in an error or a timeout (as reported by the command being executed).

4.6.1 Unpopular Servers

Table 4.2 shows the results for the unpopular Web servers. We show the average and maximum query latencies and the percentage of requests exceeding *4 seconds*. We can see that the average DNS lookup latency is around 2 seconds and that at the worst it can take over a minute to resolve a hostname. We also see that a significant number of requests takes over 4 seconds to resolve which in the context of Web browsing can induce a significant delay.

In Figure 4.10 we show the distributions of the query latencies for the unpopular servers at all three sites.

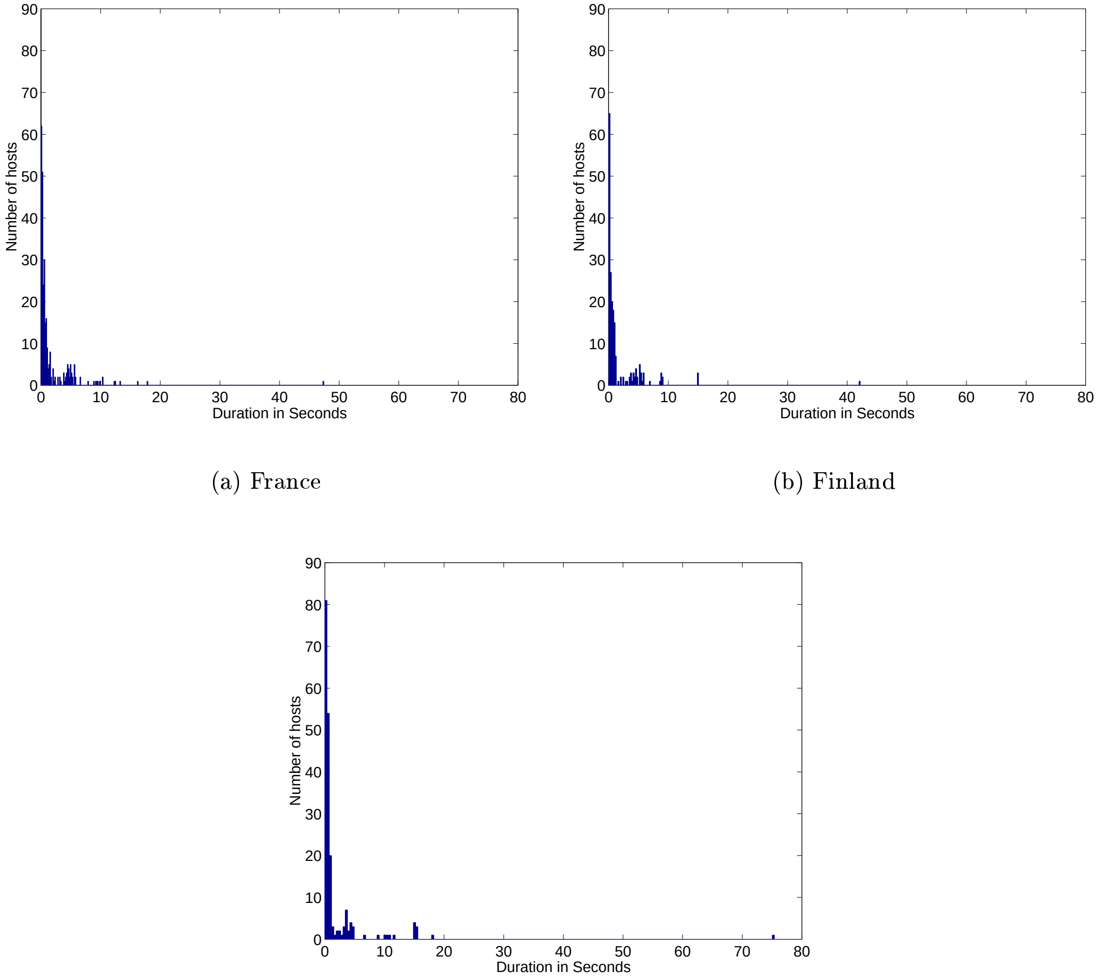
4.6.2 Popular Servers

In Table 4.3 we show the average and maximum DNS query latencies in seconds from all the three test sites as well as the percentage of queries that took longer than *2 seconds* to resolve. The last line shows the averages over all three sites. We can see that the average latencies are very low which is likely the result of the requested information being present in the local nameserver's cache. We also see that, at worst, the latency can be up to several seconds but that such events are relatively rare.

In Figure 4.11 we show the distributions of the query latencies for all the three sites.

4.6.3 Replicated Server

We also ran experiments on our local nameserver and the average time to resolve a hostname was 30 ms. The round-trip time to the national Web cache of France from our network is on the order of 20 ms, thus the total time to query a hostname at the replicated

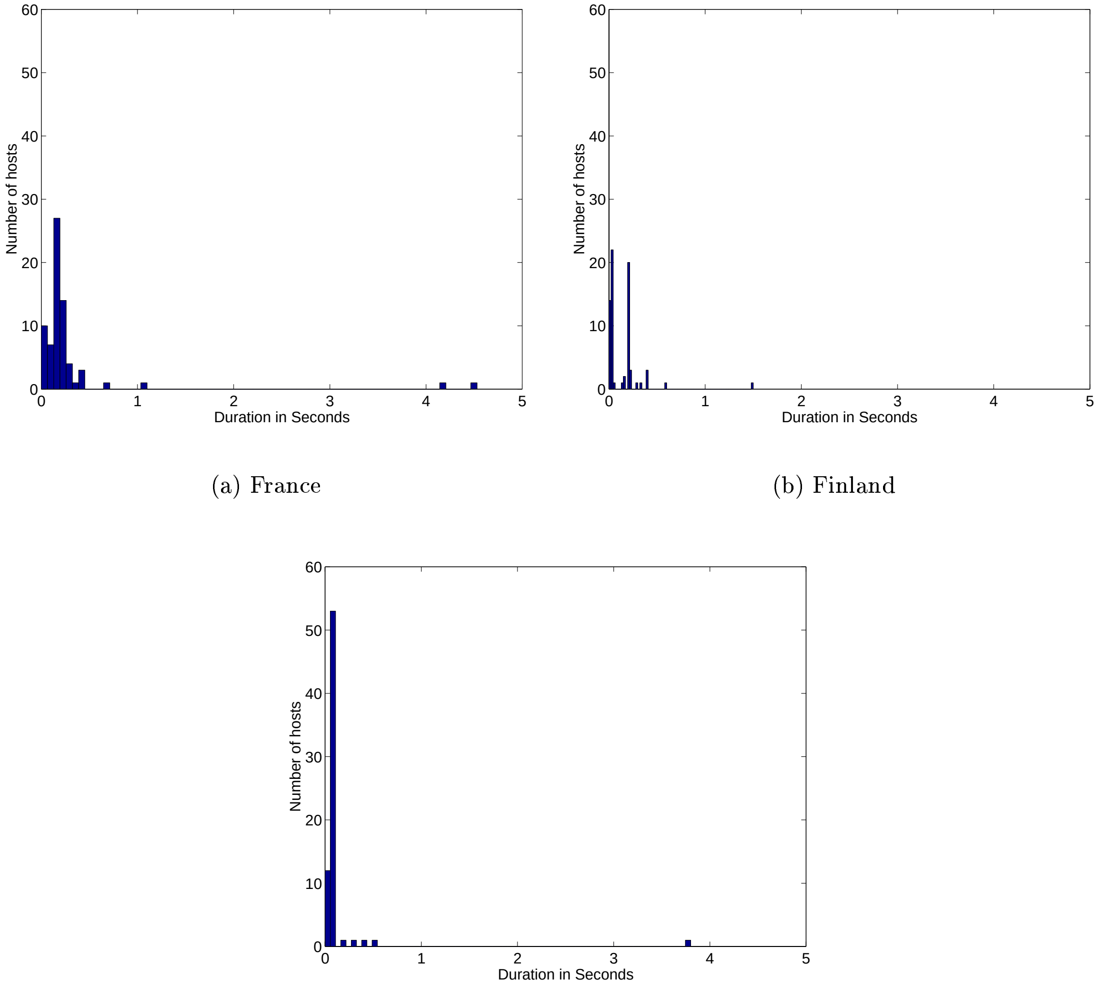


(a) France

(b) Finland

(c) US

Figure 4.10: DNS query latency histograms for unpopular servers



(a) France

(b) Finland

(c) US

Figure 4.11: DNS query latency histograms for popular servers

Site	**Average latency**	**Maximum latency**	**% of lookups exceeding 2 seconds**
FR	0.31	4.53	2.9
FI	0.14	1.5	0
US	0.22	3.8	2.9
All	0.22	4.53	1.7

Table 4.3: DNS query results for popular servers

nameserver (assuming it was placed at the same level as the national Web cache) would be on the order of 50 ms. This is a reasonable assumption since, even though the database at the replicated nameserver is larger than the one on our local nameserver, the replicated nameserver would be running on a more powerful machine (our local nameserver runs on a SPARCstation 10). If the replicated nameserver was placed closer to the client, the lookup latency would be shorter, since the round-trip time to the replicated nameserver would be much smaller.

We can conclude that the speed-up from using a replicated nameserver would be around one order of magnitude for popular hosts and two orders of magnitude for less popular hosts. This speed-up can significantly improve the Web surfing experience.

4.7 Graceful Migration from Distributed to Replicated System

Because it is unreasonable to assume that everybody would be able to upgrade their systems overnight to switch from the "legacy" DNS system to the replicated system, we need to be able to deploy the system incrementally and transparently. In this section we outline one of many possible migration strategies.

In the early stages of deployment, a relatively small number of zones could replace their secondary or primary nameservers with a replicated nameserver. Using satellite distribution or multicast IP, the replicated nameservers would share with each other the up-to-date RRs for which they are responsible. Of course, when only a subset of the local ISPs participate, a replicated nameserver will not be able to authoritatively answer queries for all hostnames. When a replicated server is unable to answer a query, it will have to resort to the legacy DNS system to obtain the RR, i.e., it will have to query a series of nameservers, as shown in Figure 4.1. Whenever a replicated nameserver obtains a RR as a result of a DNS query, it should cache that RR until the TTL of the RR has expired. Additionally, using the satellite or multicast IP infrastructure, we propose that *the replicated nameservers share with each other the RRs that they have recently requested from the legacy DNS system.* In this manner, most of the "popular" hostnames will be cached in the replicated servers, even if a given replicated nameserver handles relatively little DNS traffic. When a RR is about to expire, a replicated server can refresh the RR and then distribute it to all the other replicated servers. Note that this strategy is similar to the Sky Cache model where the participating caches pool their user communities in order to improve the performance. The advantage of this scheme is that it is easy to deploy a few replicated nameservers at the leaves of the network, but the disadvantage is that the replicated database will only contain a small (but widely requested) part of the global DNS database. In addition, only participating zones are able to distribute rapidly changing DNS information; non-participating zones would have to rely on the TTL-values to force the replicated nameservers to update their databases.

Content providers would have an incentive to place their hostnames in zones with participating ISPs (i.e., an ISP with a a replicated server), because a participating ISP will be able to quickly distribute DNS information to all other participating ISPs, even if the hosts are moved or replicated. Furthermore, users will have an incentive to subscribe to participating ISPs, since the databases in the replicated ISPs contain most of the popular DNS RRs. These incentives should provide sufficient motivation for the non-participating ISPs to become participating ISPs. Once all the ISPs participate, the legacy system (including the root servers) can be disabled, and the full benefits of the replicated DNS system can be reaped.

4.7.1 Gathering Information

If a replicated nameserver replaces a secondary nameserver for a zone, the replicated nameserver can perform zone transfers from the primary nameserver and thus obtains a copy of the resource records for that zone. Therefore the problem of gathering the DNS information reduces to getting information from zones that are not represented by a replicated nameserver. The simple solution is to contact one of the authoritative nameservers of such zones and try to perform a zone transfer. If this succeeds, the replicated nameserver can periodically perform the zone transfer in order to keep the records up-to-date. In this situation, we lose the ability to track rapidly changing information since the primary nameserver does not inform the replicated nameserver of changes. Some zones, however, do not allow other than the secondary nameservers to perform zone transfers from them. From these zones we cannot get information, except by querying separately for each resource record. In practice this means that every time the replicated nameserver receives a query for a host in such a zone, the replicated nameserver queries one of the authoritative nameservers and caches the reply like a normal local nameserver. The replicated nameserver can then refresh the resource record when it is about to expire. We will thus obtain from uncooperative zones all the resource records that clients actually request; this provides us with enough information.

This method of gathering information presents us with the problem of keeping it up-to-date. All RRs collected from uncooperative zones will have the normal DNS TTL-stamp and the replicated nameservers can store the RR until the TTL expires. If the TTL is very short, the replicated nameservers should not send that RR to the other replicated nameservers. This is in order to avoid sending an almost continuous stream of update messages which would be the result if the replicated nameserver needs to access the same RR again in the near future. Sending the updates would place an unnecessary burden on the other replicated nameservers which would have to store the RR and shortly after remove it because its TTL had expired. As a results, the replicated nameservers will contain all the RRs from the zones they are responsible for as well as the slowly changing RRs from other zones.

4.8 Security Issues and Fault Tolerance

Our architecture has one serious security hole, namely, it creates a new possibility for DNS spoofing. DNS spoofing goes as follows.

A user (`pc.client.com`) wants to access the Web servers of two competing companies, `www.company-a.com` and `www.company-b.com`. First, the client issues a DNS lookup for the address of the first server (`www.company-a.com`). The request goes to client's local nameserver, `ns.client.com`, which does not have the answer. The local nameserver contacts the nameserver of Company-A, `ns.company-a.com`, and asks for the IP-address of the Web server. Company-A's nameserver replies with the address and includes in the reply a false A-record for the Web server of Company-B, `www.company-b.com`. The reply, including the false address, is cached at the local nameserver and if the client tries to access `www.company-b.com` while the mapping is still cached, the local nameserver will return the false mapping. The client is thus redirected to whatever server the nameserver at Company-A claimed was the Web server of Company-B. This problem stems from the local nameserver's willingness to accept information that it did not ask (the IP-address of `www.company-b.com`). Modern versions of nameservers (e.g., BIND) block this security hole by not accepting RRs they have not asked.

Our architecture is vulnerable to this type of attack if replicated nameservers accept all information they receive from other nameservers (primary or replicated). In this case, the attack would be very simple. The administrator of a primary nameserver would only have to send a false mapping to the replicated nameserver and the false information would immediately be propagated to everyone on the network. If a client were to request this information, it would receive both the real information (given by the real primary nameserver) and the false information (given by the imposter). The client could easily choose to use the wrong address instead of the correct one.

To counter this problem, the replicated nameservers must verify that the RRs they receive come from a server that is authorized to provide this information, e.g., the primary nameserver of that zone. We propose the following solution. Each replicated nameserver is responsible for a zone (each of these zones may contain several DNS zones) and can only provide information for that zone. When a replicated nameserver receives updates from a primary nameserver or another replicated nameserver, it must verify that the server sending the original update is authorized to do so. This requires that we extend the DNS SOA-type RR to include the zones handled by replicated nameservers. In addition, to avoid replicated nameservers from sending RRs for non-existing zones, these extended SOA-RRs need to be signed by a trusted certification entity using the DNS security extensions [18].

4.8.1 Fault Tolerance

We must have a way of recovering from replicated nameserver crashes. These crashes present us with two problems. First, clients that were using the crashed server must be redirected to another server. Second, when the server comes back on-line, it will have a stale copy of the database and it must get a fresh copy.

To address the first problem, we propose that all local nameservers be configured with the addresses of *several* replicated nameservers. This is similar to configuring a local nameserver with the addresses of all 13 root DNS servers in modern DNS. If the client does not get a reply from a replicated nameserver even after retrying a few times, it can assume that that server has crashed and can switch to another replicated nameserver. In this situation the crash of a replicated nameserver is analogous to the crash of a root DNS server. If the replicated nameserver is acting as the local nameserver for the client, then the local clients will be without DNS service. This situation is identical to the crash of a local nameserver in modern DNS and can be handled by installing several normal local nameservers in addition to the replicated nameserver to be used as backups. These local nameservers would have the addresses of other replicated nameservers and could provide name resolution service to the clients.

The second problem, that of stale information, is more serious and more difficult to handle. When the crashed server is back on-line it must obtain a fresh copy of the database. Because the database is on the order of gigabytes, it is infeasible to download the whole database from another replicated server. We propose the following method for bringing the database up-to-date. First, all the update messages are tagged with a unique identifier (e.g., a counter) in addition to being tagged with the identity of the sender of the update message. This sender is the primary nameserver that owns the resource record, not the replicated nameserver that distributes it to the others; this is required to causally order successive modifications. When a server recovers from a crash, it knows the number and sender of the last update message it has received, and can request another replicated nameserver to send it all the update messages from that primary nameserver since that "time."

The above scheme is sufficient when the replicated server is off-line for a short period of time and we only need to reconstruct the parts that have been updated during that period. In some cases, however, this method may be costly, since the replicated nameserver has to check for updates for all the DNS zones in the world. Also, should a replicated nameserver crash so seriously that the entire database is corrupted and must be rebuilt from scratch, we propose the following strategy. In this strategy, the replicated nameserver first performs zone transfers from the primary nameservers in its zone. When the replicated nameserver receives a query for a RR it does not have, it contacts another replicated nameserver and asks for the RR. It also performs a zone transfer for the zone containing the RR *from the other replicated nameserver.* As the replicated nameserver receives updates from other replicated nameservers, it can also perform zone transfers for the zones concerned. This way, the replicated nameserver will eventually obtain a copy of all the RRs.

4.9 Conclusion

In this chapter we have presented a new design for managing the DNS database that takes advantage of recent advances in disk storage and multicast distribution technology. Our design is based on replicating the entire DNS database on geographically distributed servers, called replicated servers. Our design has two main features: (i) It is highly responsive to changes in DNS information and (ii) significantly reduces DNS lookup time. We have closely studied the issues related to storing the DNS database and evaluated the tradeoff between the staleness of DNS information and traffic load on the replicated DNS servers.

Chapter 5

Client Redirection Performance

5.1 Overview

Content distribution on the Web is moving from an architecture where objects are placed on a single, designated server to an architecture where objects are replicated on geographically distributed servers and clients transparently access a nearby copy of an object. In this chapter we study how the different redirection schemes used in modern content distribution networks affect the user-perceived performance in normal Web page viewing. Using both simulations and experiments with real Web servers we show that redirection schemes that require clients to retrieve different parts of a Web page from different servers yield suboptimal performance compared to schemes where a client accesses only one server for all the parts of a Web page. This implies that when replicating Web pages, we should treat the whole page (HTML and images) as a single entity.

5.2 Introduction

Content distribution on the Web is moving from an architecture where objects are placed on a single, designated server to an architecture where objects are replicated on geographically distributed servers and clients transparently access a nearby copy of an object [2, 3, 16, 43, 52]. The new architectures are constructed from a set of servers, which we call content servers, that contain copies of the objects. These copies can be created statically using some pre-determined rules, or dynamically on-demand depending on the load and client request patterns. When a client wants to retrieve an object, it contacts a mapping service that provides the client with an address of a content server that has a copy of the requested object. There have already been some proposals for such architectures [7, 36, 62] and several companies have started to offer dynamic content distribution services over their own networks [2, 3, 16, 43].

A vital component of a content distribution architecture is a method for redirecting clients to the content servers. What is common to most of the proposed architectures is that the client is redirected to the content server *by the system*. This means that the system must contain mechanisms for determining what is the best content server for each client. On the other hand, the proposed architectures are transparent to the client, i.e., they do not require modifying the clients or installing new software at client-side. We will discuss the details of different redirection methods in Section 5.3.

In this chapter we study how different redirection schemes affect the user-perceived performance. As the measure for performance we use the total time to download all objects on a Web page, i.e., both the HTML and embedded images. Some redirection schemes require that the client retrieves some part of a Web page (e.g., the HTML-part) from one server, and other parts (e.g., embedded images) from another server. If the client is using persistent connections of HTTP/1.1 [22], this means that the client cannot benefit from previous requests that have opened the underlying TCP-congestion window; instead the client must open a new connection to another server and this connection will initially suffer from a small congestion window. Of course, if the new server is significantly closer than the old server, the client can retrieve the remaining objects faster from the new server.

Using simulations and experiments on the Web we will evaluate the performance of different redirection strategies and how they affect the download time of the whole Web page in different situations. We will evaluate the performance of redirection strategies using both multiple parallel connections and persistent connections with pipelining.

5.2.1 Related Work

Nielsen *et al.* [56] studied the performance of persistent connections and pipelining and their results show that pipelining is essential to make persistent connections perform better than multiple, non-persistent connections in parallel. Although modern browsers implement persistent connections, they do not implement pipelining [89]. For this reason the popular browsers open several persistent connections in parallel to a server.

Recently several companies [2,3,16,43] have begun to offer content distribution services. In their services, the content is distributed over several, geographically dispersed servers and clients are directed to one of these servers using DNS redirection. We will discuss DNS redirection in more detail in Section 5.3.

Rodriguez *et al.* [69] study parallel access schemes where the client requests different parts of one object from different servers. Their scheme is designed for large objects and is not well suited for typical web page viewing; also it requires modifications to client software. In our work we study the performance of currently employed redirection schemes which redirect the client to a single server but require no modifications to client software.

This chapter is organized as follows. Section 5.3 presents the different redirection schemes used in real world systems. Section 5.4 describes the model used in our simulations and Section 5.5 presents the results obtained in the simulations. Section 5.6 presents the results obtained in experiments on the real network. Section 5.7 discusses the implications of our results. Finally, Section 5.8 concludes the chapter and presents directions for future work.

5.3 Redirection to Servers

Clients can be redirected to servers with several different methods. For example, the origin server could redirect clients using the appropriate HTTP-reply codes, the client could be given a list of alternative content servers, or the system could use other mechanisms, such as DNS redirection. These different mechanisms have all different overheads on the user-perceived performance which we will discuss in Section 5.7. For the remainder of this chapter we assume that the system uses DNS redirection (or a similar method) because of its wide-spread use in the real world.

Currently the content distribution companies redirect clients using DNS redirection in two different ways. In both redirection schemes the client sends a DNS query to the authoritative DNS server of the content distribution company which replies with an IP-address for a content server that the authoritative DNS server deems to be the best for the client. (The reply can include IP-addresses of multiple servers but modern clients use only one of them.) The client then contacts the content server and requests the object from it. The advantage of using DNS redirection is that it does not require any modifications to the client software because typically URLs identify hosts by their names.

The two different schemes are as follows. In the first scheme, which we call *full redirection*, the content distributor has complete control over the DNS mapping of the origin server. When a client requests any object from the origin server, it will get redirected to a content server. This scheme requires that either all content servers replicate all the content from the origin server, or that the content servers act as surrogate proxies for the origin server. A major advantage of full redirection is that it adapts dynamically to new hot-spots because all client requests are redirected to geographically dispersed content servers.

The second scheme, which we call *selective redirection*, goes as follows. The references to replicated objects are changed to point to a server in the content distribution network. When the client wants to retrieve a replicated object, it resolves the hostname which redirects it to a content server. In this scheme, the replicated objects appear to be simply objects that are served from another server. An advantage of this scheme is that the content servers only need to have the content that has been replicated. In modern content distribution networks that use selective redirection, the burden of deciding which objects to replicate is placed on the content provider. A system using selective redirection is slower to adapt to hot-spots because it must first identify them, change the references to the new hot objects, and possibly replicate the objects to content servers. In addition, clients that have cached references to the new hot objects (e.g., caching the HTML page referencing a hot image) would not be redirected, but would instead go to the origin server thus negating the benefits of using a content distribution network.

5.4 Simulation Model

For the simulations we used the NS network simulator [55]. We used a very simple network topology and it is shown in Figure 5.1. In Figure 5.1, C is the client, S is the server, and L is the link between the two. To represent different network conditions we varied the delay,

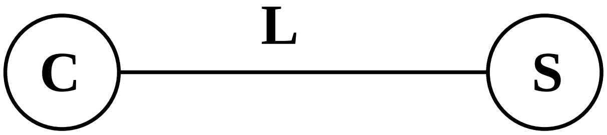


Figure 5.1: Simulation model

bandwidth, and loss rate on the link L. We used FullTCP-agents at both the client and server and we set the MSS to 1460 bytes. This is the MSS that an Ethernet-connected machine would obtain and we have obtained the same MSS on real connections to distant servers from our local Ethernet. As suggested in [56], we disabled Nagle's algorithm on the server's TCP agent.

In all of our simulations, the client first sent a request to the server, the server replied with one file (the HTML-file). When the client had received all of this file, it requested the images from the server.

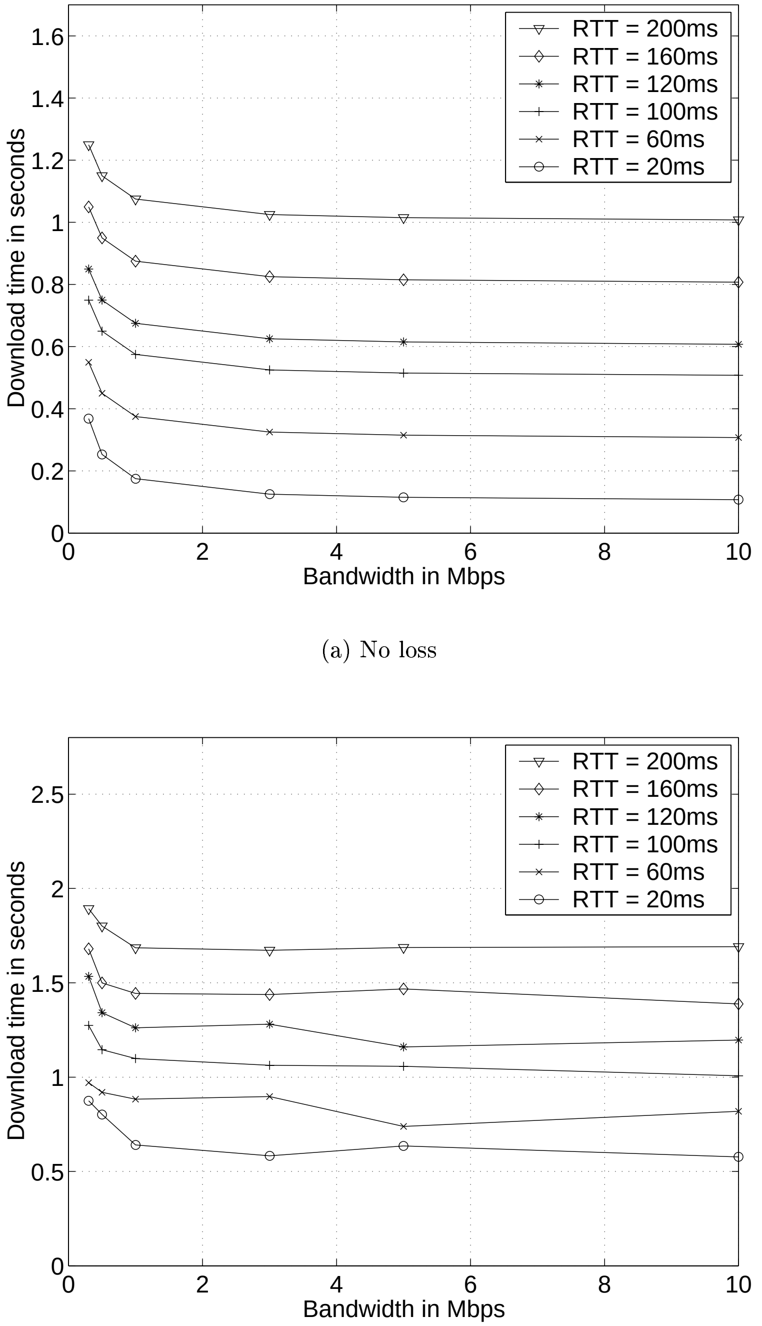
We observed that because all our HTTP-connections were short, the underlying TCP connection never progressed beyond slow-start. Therefore the bandwidth of the link had only minimal effect on the overall download time. This is also shown by the graphs in Figure 5.2 which show that the total download time stays almost constant once the bandwidth is greater than 1 Mbit/s. This is also true in the case when the loss rate on the link is high (Figure 5.2b, averaged over 3000 simulation runs).

Because of the negligible effect of the bandwidth, we compared the different redirection schemes only by varying the round-trip times and loss rates. Our simulation model does not account for server loads or how the client obtains the redirection; we will discuss these issues in Section 5.7.

5.4.1 Redirection Schemes

We compared the redirection schemes for two different clients. The first one was a client that does not implement pipelining and opens several persistent connections in parallel and the second client implemented pipelining. Given the typical number of embedded images on a page (see Section 5.4.2) and the number of persistent connections opened by popular browsers (2 or 6, see [89]), we assumed that the client opening parallel connections cannot retrieve all images with one set of connections. Instead, it first sends one batch of requests and when it receives the replies, it requests the second batch; this matches the behavior of a client implementing persistent connections without pipelining.

The baseline for our comparisons was a scheme where the client retrieves all object from the origin server. We compared this baseline scheme to two other schemes which modeled the full and selective redirections schemes. For the full redirection, we simply used a shorter round-trip time on the link to reflect the closeness of the new server. We modeled selective redirection by having the client retrieve the HTML from the origin server and being redirected to another server for the images. To model the second server, we simply used two models from Figure 5.1 in parallel and the only connection between them was



(a) No loss

(b) 5% loss rate

Figure 5.2: Effect of bandwidth on download time

Page	**HTML**	**Images**	**Parallel**
Small	5 KB	10 KB	2 KB
Medium	10 KB	20 KB	4 KB
Large	20 KB	40 KB	8 KB
X-Large	40 KB	80 KB	16 KB

Table 5.1: Different pages used in simulations

when the first model had completed its download and it triggered the second model. We used six different round-trip time values for the servers, 10, 20, 60, 100, 120, and 160 ms. We have observed that the value of 160 ms is quite typical from Europe to popular web sites in the US, and the smaller values reflect conditions within the US.

Our model does not account for the delay caused by a potential DNS lookup to get the address of the second server. This delay can be high and vary significantly [13], but it is likely to be small for popular servers. We also assume that the parallel connections do not interfere with one another and represent them by one connection which retrieves one fifth of the image data on the page. In reality, the download time of the parallel client would be dictated by the largest image because the client could not divide the images equally between all the parallel connections. Modern clients also start requesting embedded images as soon as they have seen the references to them instead of waiting for the whole HTML page to download. Our model does not take this fully into account, but the behavior of our model is appropriate for images that are referenced near the end of the HTML page.

5.4.2 Files

To estimate the size of a Web page, we downloaded all the homepages of the most popular sites from Hot100.com [30]. We found that the mean file size is 20 KB. This only includes the actual HTML for the page and does not include any embedded objects. The mean amount of embedded image data on a page is 40 KB, the mean number of embedded images on a page is 15.5, and the mean size of a single embedded image is 2.5 KB. Most of the HTML pages are at least 5 KB and almost none of the HTML pages are larger than 45 KB. To retrieve a typical homepage with all the embedded images we need to transfer around 50–60 KB and the total amount of data (HTML and images) can be as high as 250 KB. We constructed four different sized pages in order to cover as many different real pages as possible. Table 5.1 shows the different pages and the amount of HTML and image data on each of them. We also show the amount of image data retrieved by the parallel client which was equal to one fifth of the total image data.

5.5 Simulation Results

In this section we will present the results from our simulations. We ran simulations for all the different parameter values (RTT and loss rate) but we only show some of the

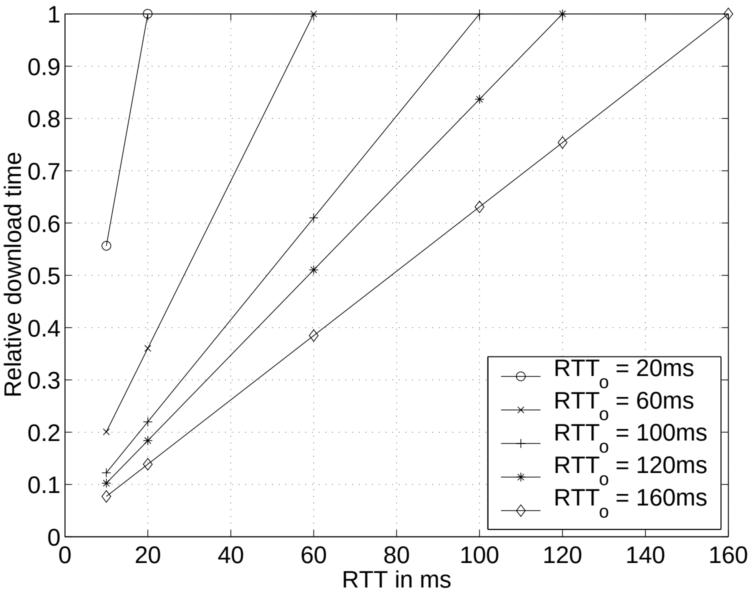


Figure 5.3: Performance of the full redirection scheme

combinations here. The results for the simulation runs not shown here were similar.

5.5.1 Loss Free Conditions

We first simulated retrievals under completely loss free conditions. In Figure 5.3 we show the performance of the full redirection scheme, i.e., retrieving everything from one server. We plot one curve for each different origin server (RTT_o), and a point on a curve shows the download time relative to the download time from the origin server if we had used a server with RTT given by the x-axis value for all the objects. For example, consider the line $RTT_o = 120\,ms$. This line compares the performance of full redirection when the RTT to the origin server is 120 ms. The points on this line show the relative download time we would have received with full redirection if the new server had had an RTT shown on the x-axis. For example, when the RTT to the new server is 60 ms, the total download time from that server would be 60% of the time it takes to complete the download from the origin server with RTT of 120 ms.

The plot shows the download times for a client using parallel connections. We observed only very small differences between different file sizes.

In Figure 5.4 we show the relative download time of the selective redirection scheme for a client using parallel connections and for the Medium and Large pages. We plot one curve for each different origin server (RTT_o). A point on a curve shows the download time relative to the baseline that a client would obtain if the origin server had a round-trip time of RTT_o and the new server had the RTT on the x-axis. The baseline in these graphs refers to retrieving all object from the origin server with round-trip time RTT_o. As we can see, typically we need the RTT to the new server to be less than 75% of RTT_o in order for it to worth it to switch to the new server. In Figure 5.5 we show the results for a client using pipelining. We can see that in this case, the RTT to the new server should be less than 50% of RTT_o.

Comparing Figures 5.4 and 5.5 we see that for parallel connections the slope of the curves is smaller, meaning that a small reduction in RTT only gets small reductions in total download time. For pipelining the slope is larger meaning larger gains for small reductions in RTT. Overall, we see that the maximum gain in download time is around 30% of the baseline. This is achieved when the new server is extremely close (RTT around 10 ms).

As we can see in Figure 5.3, if the origin server is slow (100 ms or more), we can get impressive gains if we were able to access a nearby server for all of the page. By choosing to go to a nearby server instead of the origin server for all the of the page we can download it in 10% of the time it would have taken to get the page from the origin server. The possible gains of going to a nearby fast server are much greater than the achievable gains obtained with schemes where the client must switch servers during the download. This holds for both parallel persistent connections and pipelining connections. We also see that the schemes which switch servers (Figures 5.4 and 5.5) can obtain at maximum only a 30% reduction in download time. In the same situation, by going directly to the nearby fast server, the client would reduce the download time by 90%. In other words, even if the new server would be fast enough to warrant switching to it, the client would be much better off by going to that fast server already for all the objects. We can conclude that under good network conditions, switching servers during download *will give sub-optimal performance* compared to a scheme which redirects the clients to a good server for all the objects.

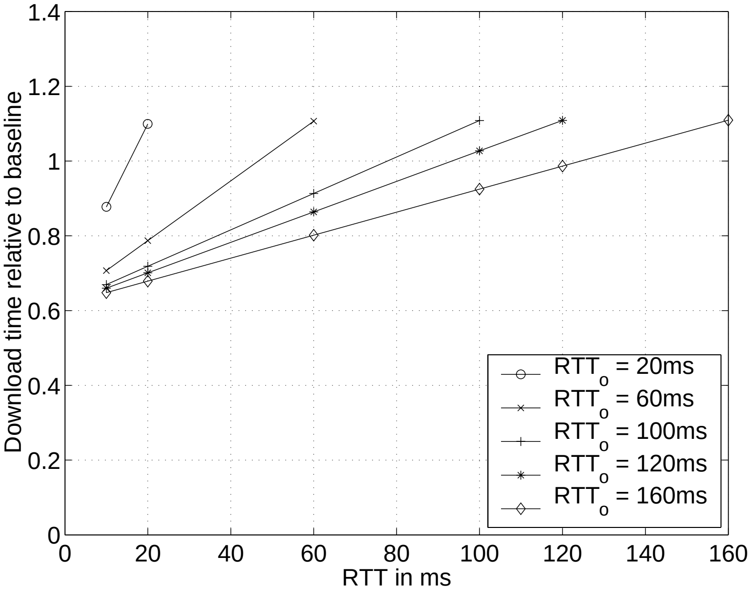
5.5.2 Simulations with Loss in Network

We then ran the same simulations using a 2% loss rate on all the links in the simulation. The results in all cases were similar to the ones obtained under loss free conditions. Figure 5.6 shows the performance of the full redirection scheme, i.e., retrieving everything from the origin server with round-trip time RTT_o. As with the no-loss situation, the differences between different file sizes were very small.

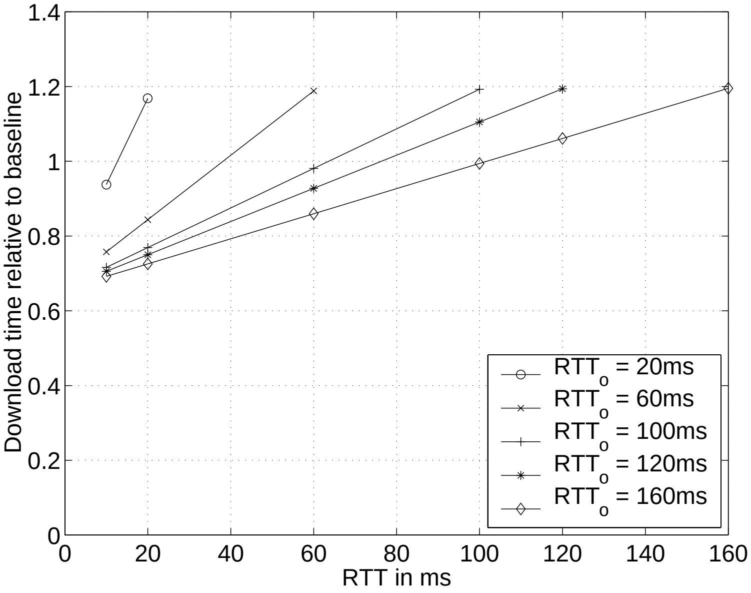
When we compare Figures 5.3 and 5.6, we see that in the no-loss situation, the maximum gains are larger than in the situation where there is loss on the link. We believe this is because the underlying TCP connection is still in slow-start and therefore its RTT-estimate has not yet adapted well enough to the link RTT. Hence, the RTT-estimates for the links with small RTTs are too high and discovering a lost packet takes more time than it would if the TCP connection knew the RTT better.

In Figure 5.7 we show the relative download times of the selective redirection scheme for a client using parallel connections and Medium and Large pages averaged over 3000 simulation runs.

As we can see, the general form of the curves matches those obtained in loss free conditions (Figures 5.4 and 5.5). The only difference is that the point where switching servers would become useful is lower than in the loss free case. Furthermore, the maximum

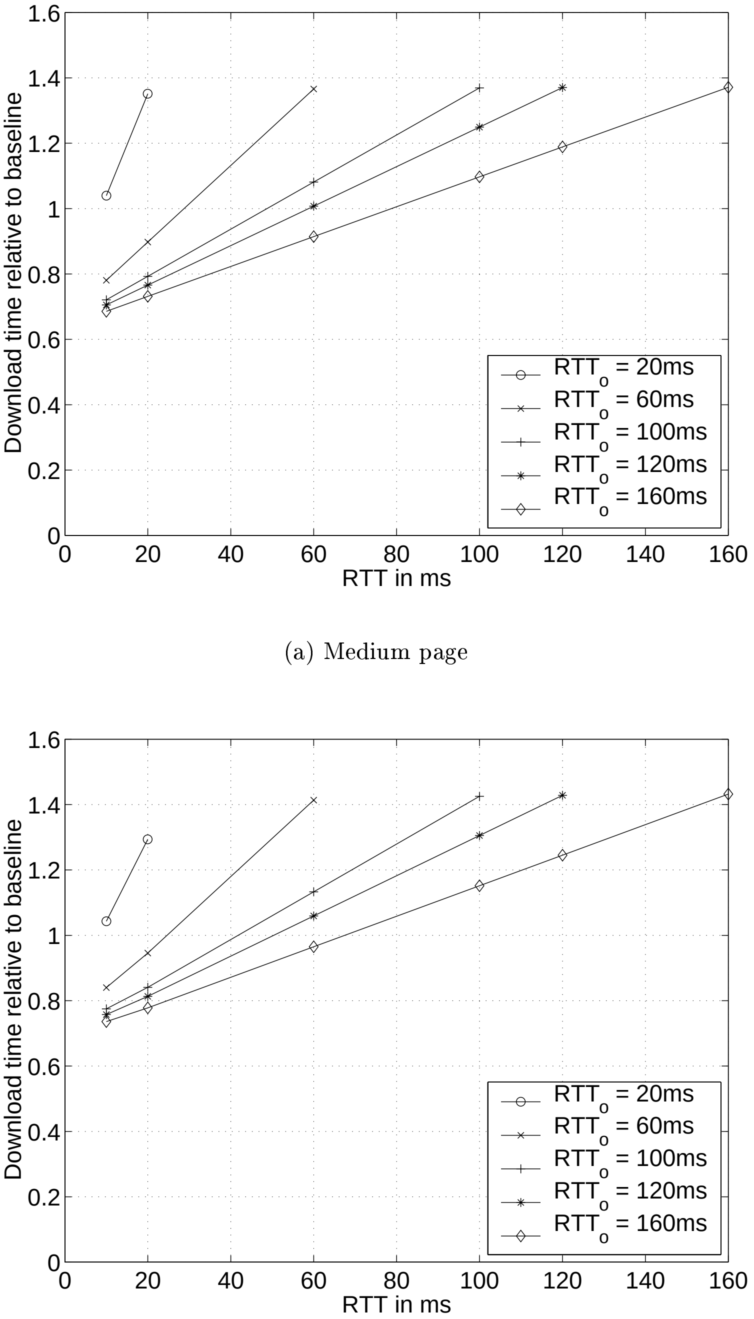


(a) Medium page



(b) Large page

Figure 5.4: Selective redirection with parallel connections



(a) Medium page

(b) Large page

Figure 5.5: Selective redirection with pipelining

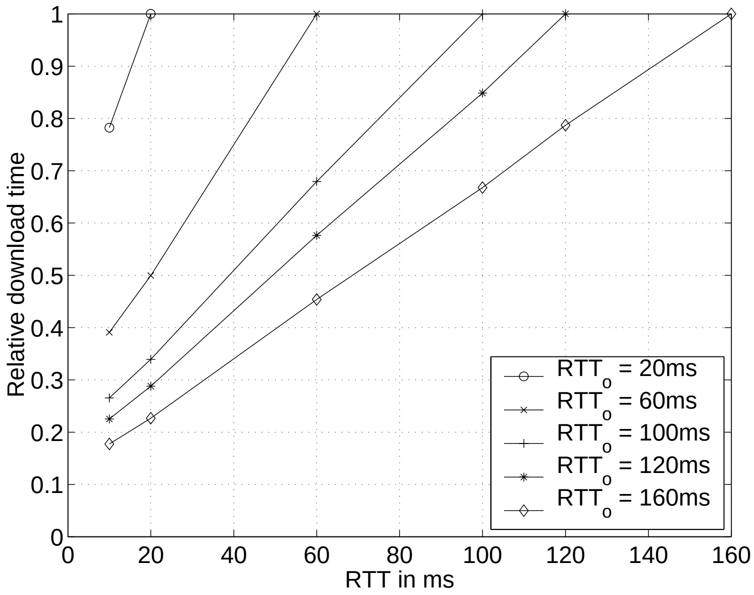


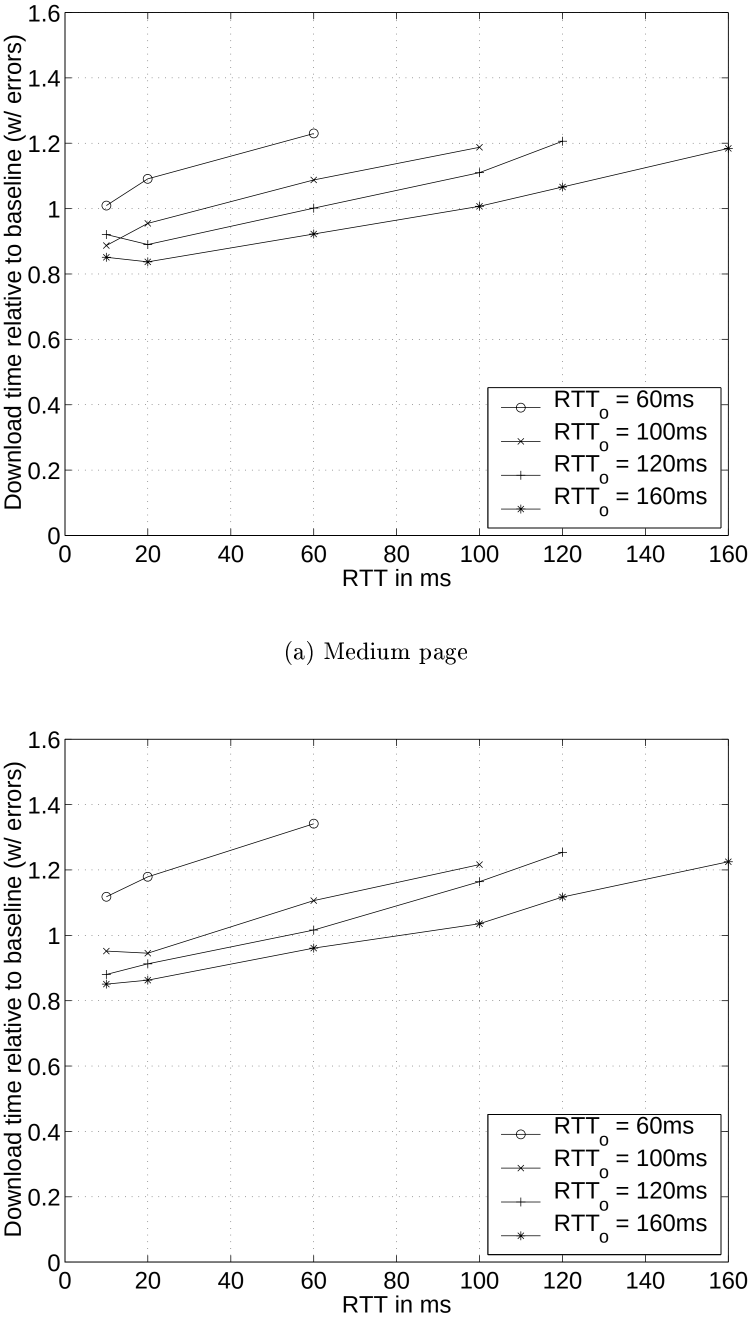
Figure 5.6: Performance of the full redirection scheme with loss

gain in download time is less than 20% and for Small pages switching servers always resulted in a slower download. We believe that the reason for the stricter RTT requirement for the new server is due to the possibility of losing TCP SYN packets when opening the connection to the new server. Most of the downloads took less than 2 seconds from start to finish, but a lost SYN packet caused a 6 second timeout. This slows down the connection to the new server considerably and gives a substantial advantage to using the persistent connection to the old server. Also, the TCP connections are in slow-start and do not therefore have an accurate estimate of the RTT and discovering lost packets will take longer on the new connection. If a packet on the persistent connection is lost, it will be discovered faster, either because of duplicate ACKs or because the TCP agents have an idea of the connection round-trip time and know when to expect packets.

The results confirm our conclusions from the loss free case. It is preferable *not to switch servers* during download of a single page. Switching servers greatly limits the gains in performance obtained from using a nearby server for all the objects. This means that *a system which forces clients to switch servers during the download of a single page will provide clients with sub-optimal service*; either the new server is not fast enough, or even if it is, the client should have been redirected to the fast server for all the objects.

5.6 Experiments

To validate the results obtained from our simulations, we ran several experiments in which we retrieved objects from real, replicated web sites. We used three different sites, Apache [6], Debian [15], and Squid [81], and chose several mirror servers of those sites for our experiments. From each of the three main sites we selected some files that matched



(a) Medium page

(b) Large page

Figure 5.7: Selective redirection with parallel connections with loss

Experiment	$\mathbf{RTT_A}$	$\mathbf{RTT_B}$	AB	BB
Apache-1	80 ms	25 ms	0.77	0.14
Apache-2	60 ms	50 ms	1.34	0.58
Debian-1	100 ms	40 ms	0.98	0.25
Debian-2	180 ms	90 ms	0.97	0.72
Debian-3	80 ms	65 ms	1.26	0.79
Squid-1	200 ms	45 ms	0.73	0.20
Squid-2	70 ms	45 ms	1.03	0.59

Table 5.2: Results from experiments

the files in our simulations as closely as possible. For Apache and Squid our files were close to the Small page in Table 5.1, and for Debian they were similar to the Medium page.

Our goal was to have our client run like a modern browser, i.e., no pipelining. We chose one HTML file and one image file from each site and divided the servers in pairs. The client would first request the both files from both servers in the pair and then request the HTML from the first server the image from the other server in the pair. Before requesting the objects we performed a DNS query on the hostnames in order to eliminate the effects of long DNS lookups. We ran our experiments several hundred times during different times of day and on several days.

We show the results from 7 of our experiments in Table 5.2. In each experiment we used a different pair of servers to get as many different combinations as possible. The RTTs to the two servers shown in columns RTT_A and RTT_B reflect typical RTT values from our client machine to the servers in the experiment. We used server A as the baseline and show the relative download times for two other download schemes in columns AB and BB. Column AB refers to experiments where the client retrieved the HTML from server A and the image data from server B. In column BB we show the relative download time when the client retrieved everything from server B.

The results we obtained in our experiments closely match those we obtained in our simulations. In fact, for all the experiments shown in Table 5.2, our simulations correctly estimated whether switching servers would result in a gain in relative download time or not.

5.7 Discussion

Our results show that the client can download a whole web page fastest if it is using persistent connections to a nearby content server (possibly using parallel persistent connections for embedded images). Of the two redirection schemes, full and selective redirection, full redirection achieves this goal easily since all requests to the origin server are redirected to a nearby content server.

With selective redirection it is possible to achieve the same effect by ensuring that all objects on a web page are replicated in the same way and thus client requests for the objects would be redirected to the same content server. This puts the burden of ensuring efficiency on the party deciding the replicated objects. Some modern systems put this responsibility on the *content provider* by allowing the content provider to tag individual objects for replication. We feel that ensuring efficient delivery of content to the clients should be the *responsibility of the content distributor*; in fact, efficient delivery of content is exactly the reason why content providers enlist the services of content distributors.

Our work is based on the assumption that the client is able to open persistent connections to all servers, although we do not assume pipelining of requests. Even though the content provider may run a web server that does not implement persistent connections, the content distributor can implement persistent connections in its own content servers. If the content provider's web server does not implement persistent connections then the RTT threshold for switching servers would be higher (because new connections to the origin server would go through slow-start again). If a suitable content server exists, the client would be better off using that server for all objects.

Our simulation model does not account for two important factors, namely server loads and redirection costs. A major reason for distributing content on several servers is to take off load from the origin server and distribute it among the content servers. Increasing the load on the origin server in our model would have the effect of making it more attractive to switch servers, i.e., it would make the RTT-threshold for switching lower. On the other hand, our model assumes that the client knows the address of the new content server. In reality, the client would have to obtain this address somehow before contacting the server. This would make switching servers less attractive, i.e., raise the RTT-threshold.

The cost of getting the redirection depends on the redirection technique used. If the system is using DNS redirection, then the client can expect to spend at least 200 ms for getting the address of the content server [35]. In the worst case, the DNS lookup can take several seconds to resolve. In our simulations and experiments all the downloads took only a few seconds at maximum and a long DNS lookup would have caused a significant slow-down for switching servers. If the client needs to contact the origin server to get the redirection, this would add at least one round-trip time to the origin server.

5.8 Conclusion

In this chapter we have evaluated the performance of the different client redirection schemes used in modern content distribution networks. Using both simulations and experiments on the real network we have found that redirection schemes, which force clients to retrieve objects on a web page from multiple servers, always yields sub-optimal performance in terms of the overall client download time compared to schemes which allow the client to retrieve all objects from one, good server. This implies that when replicating web pages, we must treat the HTML-page and the embedded images as a single entity and replicate either all or none of them. Full redirection achieves this goal and yields superior performance compared to selective redirection which may split the web page between several servers.

In our future work we will expand our simulation models to include several clients and explore the effects of different network topologies on the results. We will improve our simulation models by including other parameters, such as server load. We will also do a more extensive set of experiments to validate our conclusions.

Part II

Object Replication

Chapter 6

Replication in Content Distribution Networks

6.1 Overview

Recently the Internet has witnessed the emergence of content distribution networks (CDNs). In this chapter we study the problem of optimally replicating objects in CDN servers. In our model, each Internet Autonomous System (AS) is a node with finite storage capacity for replicating objects. The optimization problem is to replicate objects so that when clients fetch objects from the nearest CDN server with the requested object, the average number of ASs traversed is minimized. We formulate this problem as a combinatorial optimization problem. We show that this optimization problem is NP complete. We develop four natural heuristics and compare them numerically using real Internet topology data. We find that the best results are obtained with heuristics that have all the CDN servers cooperating in making the replication decisions. We also develop a model for studying the benefits of cooperation between nodes, which provides insight into peer-to-peer content distribution.

6.2 Introduction

Recently the Internet has witnessed the emergence of content distribution networks (CDNs). CDNs are targeted for speeding up the delivery of normal Web content and reduce the load on the origin servers and the network. CDNs, such as Akamai [3] or Digital Island [16], distribute content by placing it on content servers which are located near the users. A content provider can sign up for the service and have its content placed on the content servers. The content is replicated either on-demand when users request it, or it can be replicated beforehand, by pushing the content on the content servers.

In this chapter we study the problem of optimally replicating objects in CDN servers. We consider each AS as a node in a graph with one CDN server with finite storage capacity for replicating objects. The optimization problem is to replicate objects so that the average number of ASs traversed is minimized when clients fetch objects from the nearest CDN server containing the requested object. We formulate this problem as a combinatorial optimization problem and show that this optimization problem is NP complete. We develop four natural heuristics and compare them numerically using real Internet topology data. Our results show that the best results are obtained with heuristics that have all the CDN servers cooperating in making the replication decisions. We also develop a model for studying the benefits of cooperation between nodes in a peer-to-peer networking context.

This chapter is organized as follows. Section 6.3 presents the network topologies we used. In Section 6.4 we develop our cost model. Section 6.5 presents the replication heuristics we have developed and Section 6.6 presents our evaluation methods and results. Section 6.7 presents peer-to-peer content distribution and develops and evaluates a model for cooperation in peer-to-peer networks. Section 6.8 discusses related work. Finally, Section 6.9 concludes the chapter and presents directions for future work.

6.3 Network Model

Our network model is based on the actual Internet AS topology. To construct the topology, we will use data provided by NLANR [57]. This data represents summaries of Internet routing data collected in the Route Views Project [74] from 1997 to beginning of 2000. These summaries provide information about which ASs are connected to each other. From these summaries we constructed a graph where the nodes are the ASs and the edges are the inter-AS connections. We then calculated the shortest paths between all the node pairs, and used this data to form a distance matrix for the network.

As discussed in [25], this method is slightly inaccurate because we cannot infer that all the connections are valid. For example, consider a small ISP that buys connectivity from two bigger ISPs. For each of the providers, our graph will show an edge between the small ISP and the provider, but in reality the small ISP would not route traffic from one of its providers to another. This is not likely to be a problem, though, since in most cases the two providers in this case either have a direct link between them or are able to connect via a common provider.

A CDN tries to put its content servers as close to users as possible. By placing the clients and the content servers (storage nodes in our network) in the same nodes, we can simulate the ideal redirection where all clients are always redirected to the closest server (the redirection in this case is static). By studying different replication strategies we can determine which strategies a CDN should use when deciding which objects to replicate. Because a CDN has complete knowledge of its network, we can use strategies which require global information or cooperation.

6.3.1 Network Topologies

We downloaded several different files from NLANR [57], each describing the AS topology on a different day. Table 6.1 shows the different topologies we used, the number of ASs in each one, the number of leaf ASs, the average length of the shortest path between any two nodes, and the average distance of the shortest path between any two leaf nodes.

Date	**Number of nodes**	**Number of leaf nodes**	**Average distance**	**Avg. leaf distance**
97/12/21	3184	1423	3.76	4.34
99/01/11	549	136	3.40	4.18
99/12/08	767	222	3.03	3.38
99/12/11	1477	502	3.45	4.10

Table 6.1: AS topologies

From topologies in Table 6.1 and other topologies we downloaded (not shown), we see that the average distance between nodes tends to increase as the number of nodes in the network increases. It is also important to note that these topologies *do not* contain all of the ASs in the Internet, but only a subset of them. Therefore, the full Internet AS topology would likely have a higher average distance than any of the topologies we used.

6.4 Cost Model

We consider a network where the nodes are the autonomous systems (ASs). For simplicity, we assume that one AS is the same as one ISP. We have I ASs in the network. AS i, $i \in 1, 2, \ldots, I$, has S_i bytes of storage capacity and has clients that request objects at aggregate rate λ_i.

We have J objects. Object j has a size of b_j, $j \in \{1, 2, \ldots, J\}$ and a request probability p_j which is the probability that a client will request this object. We assume that client request patterns are homogeneous, i.e., the p_j's are the same for all ASs. But this model can be extended to include other request patterns by using p_{ij}, which would be the request probability for object j from AS i.

We have the following variables:

$$x_{ij} = \begin{cases} 1 & \text{if object } j \text{ is stored at AS } i, \\ 0 & \text{otherwise.} \end{cases}$$

The storage is constrained by the space available at AS i, that is

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i \qquad i = 1, \ldots, I$$

The goal is to choose the x_{ij}'s so that a given performance metric is minimized. In this chapter our goal is to minimize the average number of inter-AS hops that a request must traverse. This reflects the download time of an object to some degree and can thus be used as an indicator of the user perceived latency.

We denote the matrix of all x_{ij}'s by $\mathbf{x}$. Furthermore, we assume that each object j is initially placed on an origin server; we denote by O_j the AS that contains this origin server.

We assume that all of the objects are always available in their origin servers, regardless of the placement $\mathbf{x}$. We denote the placement of objects to origin servers as $\mathbf{x}_o$. Note that this origin server placement does not count against the storage capacity S_{O_j}.

The average number of hops that a request must traverse from AS i is

$$C_i(\mathbf{x}) = \sum_{j=1}^{J} p_j d_{ij}(\mathbf{x}) \tag{6.1}$$

where $d_{ij}(\mathbf{x})$ is the shortest distance to a copy of object j from AS i under the placement $\mathbf{x}$. This nearest copy is either in the origin AS O_j, or in another AS where the object has been replicated. We assume that the client is always redirected to the nearest copy. In this chapter we do not consider the mechanisms used to redirect clients, but instead assume that such a mechanism is in place.

Let $\Lambda = \sum_i \lambda_i$ be the total request rate of all ASs. The average number of hops from all ASs is then

$$\begin{aligned} C(\mathbf{x}) &= \frac{1}{\Lambda} \sum_{i=1}^{I} \lambda_i C_i(\mathbf{x}) \\ &= \frac{1}{\Lambda} \sum_{i=1}^{I} \sum_{j=1}^{J} \lambda_i p_j d_{ij}(\mathbf{x}) \\ &= \sum_{i=1}^{I} \sum_{j=1}^{J} s_{ij} d_{ij}(\mathbf{x}) \qquad (6.2) \end{aligned}$$

where $s_{ij} = \lambda_i p_j / \Lambda$. The placement $\mathbf{x}$ is subject to

$$\sum_{j=1}^{J} b_j x_{ij} \le S_i \qquad i = 1, \ldots, I$$

This cost function represents the long term average cost. For a large number of objects and ASs, it is not feasible to solve this problem optimally; in fact, as we show in the next section, this problem is NP-complete.

6.4.1 Proving NP-Completeness

In order to prove that our optimization problem is NP-complete, we first formulate the problem as a decision problem. Given a target number of hops T, we ask is there a placement $\mathbf{x}$ such that

$$\sum_{i=1}^{I} \sum_{j=1}^{J} s_{ij} d_{ij}(\mathbf{x}) \le T$$

subject to

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i \qquad i = 1, \ldots, I$$

We prove the NP-completeness of this problem by showing that it belongs in NP and then we reduce the knapsack problem to a special case of our problem. This proves the NP-completeness.

The problem is easily seen to be in NP. Given a placement $\mathbf{x}$ and number of hops T, we can verify in polynomial time whether the placement results in an average cost of less than T hops.

Next, we consider the special case where $S_1 = S$, $S_i = 0$, $i = 2, \ldots, I$, $\lambda_i = \lambda$, $i = 1, \ldots, I$, $p_j = p$, $j = 1, \ldots, J$, i.e., we have only one AS on which to place objects, all ASs have the same request rate, and all objects are equally popular.

Recall that each object j is always available at the origin AS O_j. The cost of getting object j for a client in AS i is $d_{ij}(\mathbf{x}_o)$. Because all clients always go to the nearest copy, placing copies on the only available AS can only decrease the cost for any client, i.e., $d_{ij}(\mathbf{x}) \leq d_{ij}(\mathbf{x}_o)$ for all i and j. We define the utility of placing object j on AS i as $u(j) = \sum_i [d_{ij}(\mathbf{x}_o) - d_{ij}(\mathbf{x})]$, i.e., the decrease in number of hops we would obtain if we placed object j in the AS.

Given target decrease in number of hops T' we now ask if there is a set of objects J' such that

$$\sum_{j \in J'} b_j \leq S \quad \text{and} \quad \sum_{j \in J'} u(j) \geq T'$$

This problem is identical to the well-known NP-complete knapsack problem [26]. Given that our placement problem belongs in NP and that the knapsack problem reduces to it, we know that our placement problem is NP-complete.

6.5 Replication Heuristics

Because our optimization problem is NP-complete, finding the optimal solution is not feasible. Therefore we have designed several heuristics that use the available information in different ways in order to get the best results.

In our simulations we use the following heuristics.

1. **Random**. Assigns objects to storage nodes randomly subject to the storage constraints. We pick one object with uniform probability and one node with uniform probability, and we store the object in that node. If the node already stores that object, we pick a new object and a new node. As a result, an object can be assigned to several nodes, but a node will have at maximum one copy of an object.

2. **Popularity**. Each node stores the most popular objects among its clients. The node sorts the objects in decreasing order of popularity and stores as many objects in this order as the storage constraint allows. The node can estimate the popularities by observing the requests it receives form its clients. This heuristic does not require the node to get any information from outside of the node. Note that in our case, the object popularities p_j are the same across all nodes, hence all the nodes will store the objects in the same order but subject to different storage constraints.

3. **Greedy-Single**. Each node i calculates $C_{ij} = p_j d_{ij}(\mathbf{x}_o)$ for each object j. This represents the contribution of an individual object to (6.1) under the initial placement. The node then sorts the objects in decreasing order of C_{ij} and stores as many objects in this order as the storage constraint allows. The popularities are obtained as in the *Popularity* heuristics, but the CDN also needs information about the network topology in order to estimate the d_{ij}'s. Note that the C_{ij}'s are calculated only once under the placement $\mathbf{x}_o$ and are not adjusted when objects are stored. This means that every node stores objects independently of all the other nodes and no cooperation between nodes is required.

4. **Greedy-Global** The CDN first calculates $C_{ij} = \lambda_i p_j d_{ij}(\mathbf{x}_o)$ for all nodes i and objects j. Then the CDN picks the node-object-pair which has the highest C_{ij} and stores that object in that node. This results in a new placement $\mathbf{x}_1$. Then the CDN re-calculates the costs C_{ij} under the new placement and pick the node-object-pair that has the highest cost. We store that object in that node and obtain a new placement $\mathbf{x}_2$. We iterate this until all the storage nodes have been filled.

We do not consider any variants of these base heuristics, such as using popularity divided by object size, but evaluate only the performance of the base heuristics. Different variants of these heuristics have been studied in the context of web caching (see [33] and references therein) and any such improvements could be used directly to enhance the performance of our heuristics.

6.6 Evaluation of Heuristics

We evaluated the performances of our heuristics using the topologies from Section 6.3.1. We ran each heuristic on each topology using different parameters for object popularity and storage capacity in the nodes.

We assigned the popularities to the objects from a Zipf-like distribution where we varied the parameter from 0.6 to 1.0. Values between 0.6 and 0.8 have been typically observed in Web proxy traffic [10]. Much higher values, up to 1.4, have also been discovered in the context of popular Web servers [59]. Object sizes were randomly drawn from a uniform distribution. The storage capacity at each node was fixed to some percentage of the total size of the set of objects. We varied this percentage in the course of the experiments. All the client nodes had the same request rate λ.

We placed all the content servers at the leafs of the network, i.e., for all non-leaf ASs we set $S_i = 0$. Even though this assignment of storage capacity is artificial, it allows us to study the performances of our heuristics better. By placing all the storage capacity and objects at the edges, the average number of hops needed to obtain an object is higher than with a more realistic assignment. This makes it more important to replicate the right objects and lets us see more clearly the differences between our heuristics.

The baseline for our experiments was the initial placement $\mathbf{x}_o$ which we obtained by randomly assigning objects to storage nodes. We compared the performance of each of the heuristics to this baseline and report the relative performance obtained with each heuristic. Because the memory requirements for the experiments grow with the product IJ, we were not able to run all experiments for all the topologies.

Figure 6.1 shows the results from experiments with 1000 objects. We only show results for two topologies, but we observed that the results for the remaining two were similar to the two shown here. On the x-axis we plot the amount of storage at a node as percentage of the total size of the objects. On the y-axis we plot the performance relative to the baseline. In each graph, we plot different curves for different heuristics and different values of the Zipf-parameter (0.6, and 1.0). Note that we only plot the *Random* heuristic with Zipf-parameter 1.0. The performance of the *Random* heuristic was similar for the other Zipf-values. The curves of the other heuristics for Zipf-values between 0.6 and 1.0 was between the curves plotted.

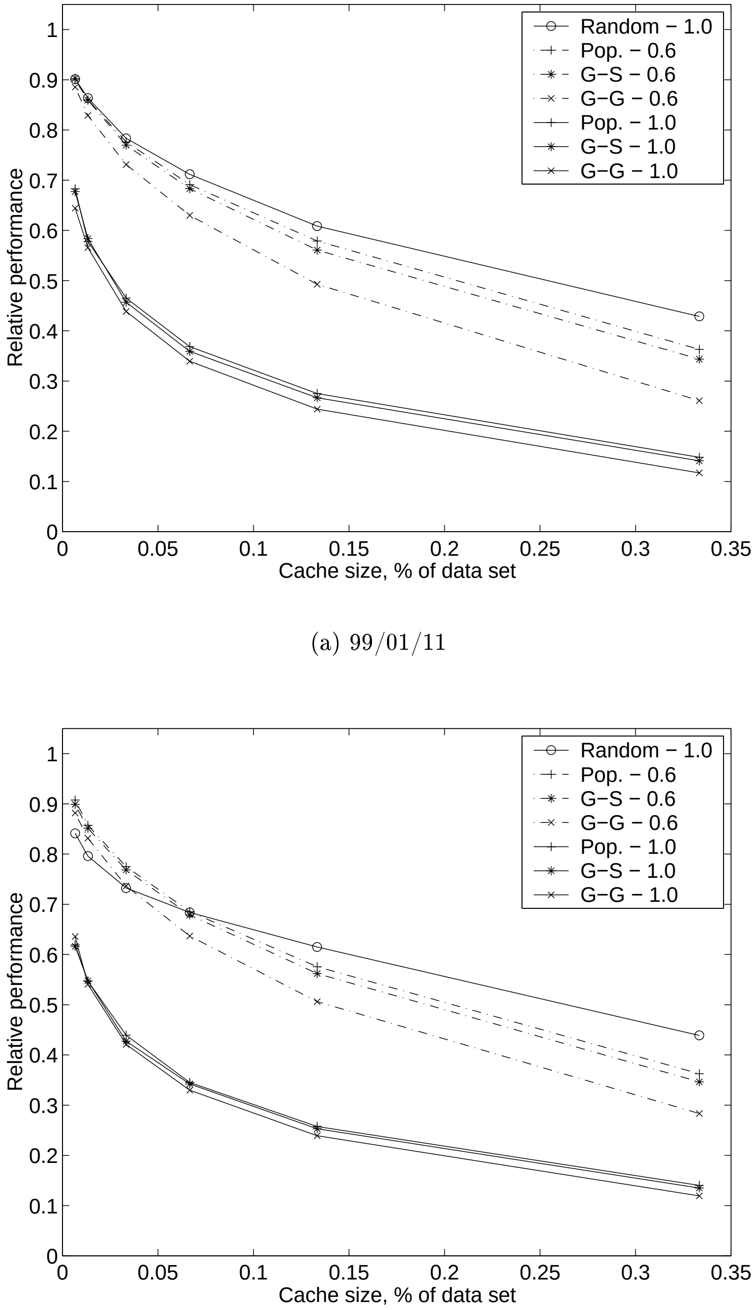
From Figure 6.1 we can see that *Greedy-Global* is the best performing heuristic. The second best is *Greedy-Single*, followed closely by *Popularity*. *Random* heuristic is consistently the worst and does not achieve substantial reductions in number of hops, even for large storage capacities. As we mentioned, the performance of *Random* did not change much with different Zipf-parameter values.

As Figure 6.1 shows, the gains increase logarithmically with increased storage capacity. The main determining factor is the Zipf-parameter value. The larger this value is, the smaller is the number of objects generating a large amount of requests. Thus, it is easy to significantly reduce the cost if only a small number of objects is very popular. To reduce the number of hops by 50%, we need only a small amount of storage for parameter 1.0 and up to 25% of total data set for parameter value 0.6.

In Figure 6.2 we plot the results from experiments with 10,000 objects. Due to memory limitations, we were not able to run *Greedy-Global* for all the topologies, therefore it is not shown on the plots.

The results are very similar to the results from the previous experiment. *Random* is still the worst in terms of performance, but the difference between *Popularity* and *Greedy-Single* has decreased. This is because as the number of objects grows, the individual object popularities will get smaller. Therefore, the contribution of an individual object's retrieval cost to the global cost (6.2) will get proportionally smaller. Hence, the popularity of the object becomes more important in determining the cost. The product of popularity and distance used by *Greedy-Single* is still a slightly better indicator, but the difference is minimal.

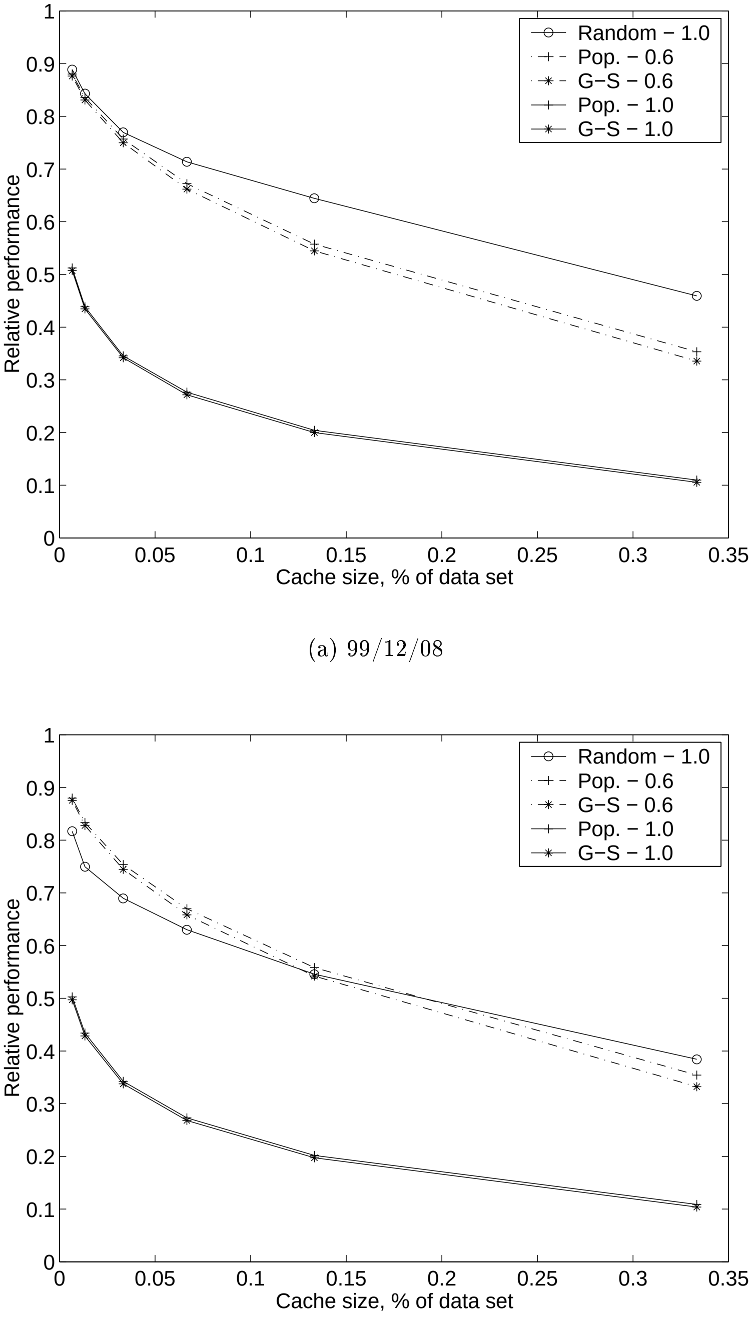
From our results we can conclude that the best performance is obtained when the



(a) 99/01/11

(b) 99/12/08

Figure 6.1: Experiments with 1,000 objects



(a) 99/12/08

(b) 99/12/11

Figure 6.2: Experiments with 10,000 objects

object replication is coordinated by a single source, in our case, this would be the CDN. The difference in performance between *Greedy-Global* and the other two heuristics is quite significant, especially for small Zipf-parameter values. In Figure 6.1 we can see that the largest improvements in performance are up to 24%. This result shows that *a CDN should use a coordinated replication strategy* and not let the CDN servers act on their own.

6.7 Peer-to-Peer Content Distribution

We now introduce a new form of content distribution, namely peer-to-peer content distribution. Peer-to-peer networks have recently emerged as a new form of content distribution and are mainly used to share individual files between users. In peer-to-peer networks, such as Napster [51], or Gnutella [28], individual users decide to share files with others. With the help of a directory service, users can determine where different files can be downloaded from. While this new model does not directly compete with the traditional Web browsing model, it is establishing itself as a means for distributing larger files, such as applications or music, between users. Because peer-to-peer networks are made up of individual users, we cannot use strategies which require global information and coordination of nodes as with CDNs. Instead we must restrict ourselves to strategies that need only locally available information; one example of such strategy is the *Popularity* heuristic.

As before, we assume that one AS in our network corresponds to one ISP. We can view the storage capacity in an AS as the aggregation of the peer-to-peer storage offered by the users in that AS. In a similar vein, $x_{ij} = 1$ means that at least one user in AS i has a copy of object j. We assume that the cost of retrieving objects from other users in the same AS is negligible; this is consistent with our definition of retrieval cost in Section 6.4. In this section we will investigate the benefits of cooperation between the users in different ASs. Our goal is to see if the users in a peer-to-peer network could gain anything from cooperating with other nearby users.

We will now develop a cooperation model for peer-to-peer content distribution under the popularity heuristic. For simplicity and ease of notation, we shall assume throughout this section that all objects have the same size, i.e., $b_j = b$ for $j = 1, \ldots, J$.

Consider two ASs, A and B. Denote the shortest path between them by D_{AB}. Let K be the number of objects that each AS can store. We do not assume anything about the relationship between the two ASs, except that the distance between them is D_{AB}. In particular, we do not assume that one is the access provider of the other one. We assume that $\lambda_A = \lambda_B$.

If both ASs act independently, they would both cache the K most popular objects and the average number of hops for requests from A and B would be

$$\begin{aligned}&\frac{\lambda_A}{\lambda_A + \lambda_B} \sum_{j=K+1}^{J} p_j d_{Aj} + \frac{\lambda_A}{\lambda_A + \lambda_B} \sum_{j=K+1}^{J} p_j d_{Bj} \\ =&\frac{1}{2} \sum_{j=K+1}^{J} p_j d_{Aj} + \frac{1}{2} \sum_{j=K+1}^{J} p_j d_{Bj} \qquad (6.3)\end{aligned}$$

We now consider the case where the two ASs cooperate and do not necessarily both store the same objects. We assume that both of them store a copy of the L most popular objects ($L \leq K$), and that in addition A stores objects $(L+1), \ldots, K$, and B stores objects $(K+1), \ldots, (2K-L)$. Because the request rates are identical, it does not matter how the objects $(L+1), \ldots, (2K-L)$ are shared between A and B. Note that because of the same reason, we only need to consider replicating objects in A and B and not in the intermediate nodes.

The average number of hops for requests from A and B under this scheme is

$$\begin{aligned} &\frac{1}{2}\sum_{j=K+1}^{2K-L} p_j D_{AB} + \frac{1}{2}\sum_{j=2K-L+1}^{J} p_j d_{Aj} \\ &\quad + \frac{1}{2}\sum_{j=L+1}^{K} p_j D_{AB} + \frac{1}{2}\sum_{j=2K-L+1}^{J} p_j d_{Bj} \\ = &\frac{1}{2}\sum_{j=L+1}^{2K-L} p_j D_{AB} + \frac{1}{2}\sum_{j=2K-L+1}^{J} (p_j d_{Aj} + p_j d_{Bj}) \end{aligned} \tag{6.4}$$

The difference between (6.3) and (6.4) is

$$\frac{1}{2}\sum_{j=K+1}^{2K-L} (p_j d_{Aj} + p_j d_{Bj}) - \frac{1}{2}\sum_{j=L+1}^{2K-L} p_j D_{AB} \tag{6.5}$$

If (6.5) is greater than zero, then it is better for A and B to cooperate. By assuming $d_{Aj} = d_{Bj} = d_{avg}$, we can calculate the value of d_{avg} where cooperation becomes the preferred strategy. The equation becomes

$$\frac{1}{2}\sum_{j=K+1}^{2K-L} (p_j d_{avg} + p_j d_{avg}) - \frac{1}{2}\sum_{j=L+1}^{2K-L} p_j D_{AB} > 0 \tag{6.6}$$

Solving for d_{avg}, we get

$$d_{avg} > \frac{D_{AB}}{2} \frac{\sum_{j=L+1}^{2K-L} p_j}{\sum_{j=K+1}^{2K-L} p_j} \tag{6.7}$$

If (6.7) holds, then it is better for A and B to cooperate. Because d_{avg} is likely to be reasonably stable over long intervals and because p_j's are known to both parties, A and B can use (6.7) as a quick test to see whether they could gain anything by cooperating. For the test, A and B either need to specify the value of L or verify equation (6.7) over several values of L.

In the following, we will consider two values for D_{AB}, namely 1 and 2. If D_{AB} is equal to 1, then A and B are neighbors. We plotted (6.5) for several Zipf-parameter values, average distances d_{avg}, and values of K, and in all cases, cooperation is almost always superior to a non-cooperating strategy. In some instances, however, the difference between the two is small.

In Figure 6.3 we show typical plots of (6.5). In these plots, we have 1,000 objects available and both nodes have capacity to store 50 objects, or 5% of the data set. We show two plots for two different values of the Zipf-distribution parameter. On the x-axis we plot the value of L, i.e., the number of objects stored at both A and B, and on the y-axis we show the value of (6.5), that is, the difference between individual and cooperative strategies in hops. If the difference is positive, then cooperation is better.

As we can see from the plots in Figure 6.3, there are always some values of L for which cooperation gives better results, but in some cases the gains are small. We can see that as d_{avg} gets smaller, the gains become smaller. This is no surprise since a small d_{avg} means that the requested objects are typically already very close and cooperation would not help much. We also see that as d_{avg} increases, the potential gains increase significantly.

Even though a gain of 0.2 hops may not seem much, it is important to note that the difference between *Popularity* heuristic and *Greedy-Single* was typically less than 0.05 hops. The difference between *Popularity* and *Greedy-Global* was 0.1–0.15 hops. Therefore, two cooperating nodes using the *Popularity* heuristic would obtain significantly better performance than they could hope to obtain by acting independently. We can conclude that *cooperation is much more efficient at reducing the cost* than changing a heuristic from a popularity based heuristic into a greedy one.

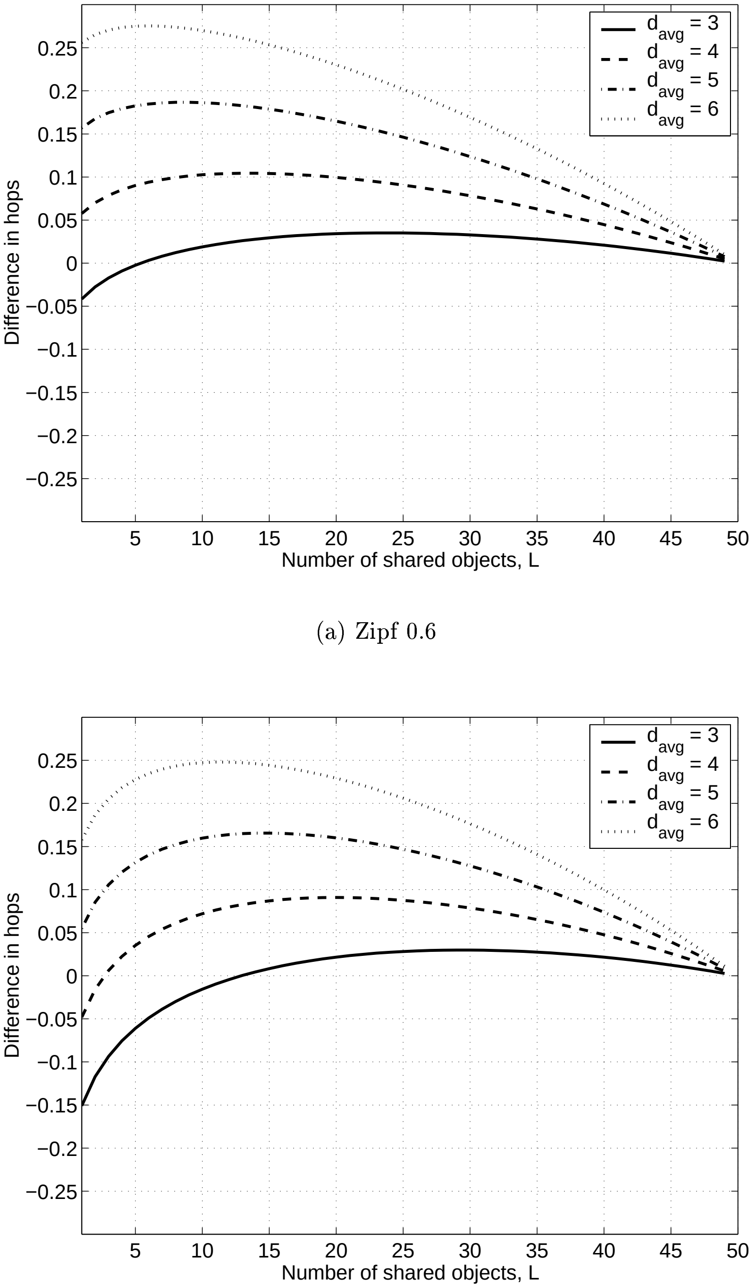
If D_{AB} is equal to 2, then A and B are separated by one AS. One example of this case is shown in Figure 6.4 where we have three nodes, one parent and two children. In this case, the cooperation would happen between the children. Figure 6.5 shows the graphs for this case.

Comparing Figures 6.3 and 6.5 we can see that the gains get smaller as D_{AB} increases. When D_{AB} increases even further, cooperation will no longer be beneficial to A and B. This is to be expected, since in the network topologies we used, the average distance between leaf nodes was around 4 and therefore the distance between A and B would be roughly the same as the distance to the origin server.

We also investigated cases where there are a very large number of objects in the network. In these cases, the general form of the gains matches those in Figures 6.3 and 6.5, but the actual gain is slightly lower. This is because the storage nodes can only hold a very small fraction of the objects, and even two nodes co-operating cannot hold enough objects to reduce the average number of hops significantly. However, even in these cases, cooperation yielded a smaller average number of hops than not cooperating. For example, with 1 million objects and K equal to 2000, the maximum gain was around 0.2 hops as opposed to 0.27 with 1000 objects and $K = 50$ (for $d_{avg} = 6$).

6.8 Related Work

Related work on replicating content has mostly concentrated on the problem of placing the replica servers for *one origin server*. In this chapter we consider a more global case



(a) Zipf 0.6

(b) Zipf 0.8

Figure 6.3: Gain from cooperation for $K = 50$, 1000 objects

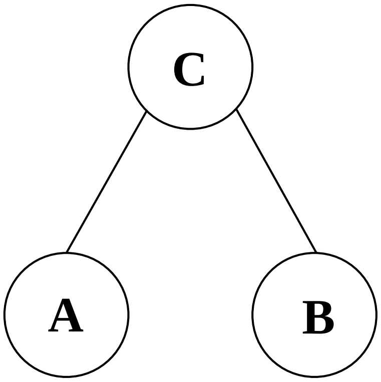


Figure 6.4: 3-way cooperation

where we have content from several origin servers and we decide which objects to replicate on the replica servers.

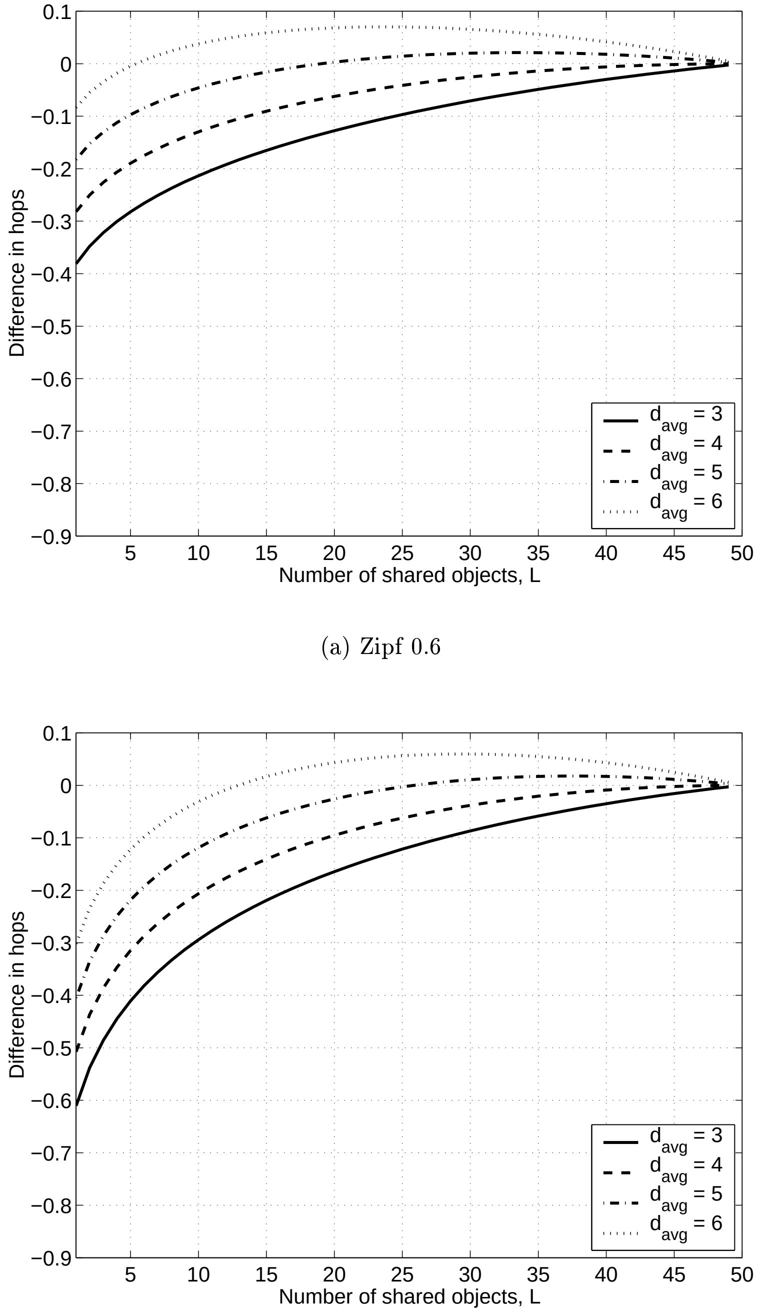
In both [41] and [12], the authors consider the problem of placing replicas for one origin server on a tree topology. While the real Internet topology is not a tree, this simpler approach allows the authors to develop optimal algorithms. The tree-approach may not generalize, because it would require that the trees for different origin servers overlap at the replica sites which cannot be guaranteed without a manual selection of the replica sites. Also, in [41], the algorithm is of a high computational complexity $O(N^3M^2)$.

In [61] the authors present their algorithms for placing server replicas in a CDN. They assume that the replicas are complete replicas and they do not study replicating individual objects; in our work we make replication decisions on a per-object granularity. They formulate the problem as the NP-complete K-median problem, develop heuristics and evaluate their performance. They only consider placing replicas for a single origin server; our heuristics replicate objects from all the origin servers. All of their heuristics require information about the network topology as well as client request loads. Also, in their work, all the replicas act independently and they do not study any cooperating schemes.

In [38] the authors consider the placement of intercepting proxies inside the network to reduce the download time. They present optimal solutions for simple topologies, such as line and ring, and consider the case of placing proxies for a single server in a tree topology. Relying on intercepting proxies requires that routing is stable during the lifetime of the connection.

6.9 Conclusion

In this chapter we have studied the problem of optimally replicating objects in CDN servers. We treat each AS as a node with finite capacity for storing objects. Our optimization problem is to replicate objects so that when clients fetch objects from the nearest CDN server, the average number of ASs traversed is minimized. We have formulated this problem as a combinatorial optimization problem and have shown it to be NP complete.



(a) Zipf 0.6

(b) Zipf 0.8

Figure 6.5: Gain from cooperation for $D_{AB} = 2$, $K = 50$, 1000 objects

We have developed four natural heuristics and compared them numerically using real Internet topology data. Our results show that the best performing heuristic is *Greedy-Global* which has all the CDN servers cooperating. The difference in performance between *Greedy-Global* and the simpler heuristics was up to 24%.

We have also studied peer-to-peer content distribution and developed a model for studying the benefits of cooperation between nodes. Our evaluation of the cooperation model shows that nodes using simple heuristics and intelligent cooperation can get significant performance gains.

The field of content distribution and peer-to-peer networks has significant potential for future research. Our future work will look more closely into inter-node cooperation and investigate the optimization problem more closely in order to establish lower bounds on the achievable performance. We also plan to extend our cost model to include other important factors, such as network traffic or server load.

Chapter 7

Replication in Peer-to-Peer Communities

7.1 Overview

A peer is a host that intermittently connects to the Internet, typically changing its IP address at each new connection. In a P2P file-sharing community, each peer dedicates a fraction of its disk storage and access bandwidth to the community. We argue that the performance of peer communities can be significantly enhanced if the peer community carefully coordinates the replication of its content. We distinguish between two types of coordinated peer communities: content caches, for which content can be retrieved from outside the community if unavailable from within the community; and peer clubs, for which all content must be retrieved from inside the community. For both peer caches and clubs, we study the optimal replication of content. We formulate an integer programming problem that provides the exact optimal (albeit academic) solution. We then provide a number of adaptive, distributed algorithms for replicating the content on-the-fly. Our algorithms combined with a distributed least-frequently-used (LFU) replacement policy are shown to provide near-optimal replication. We also provide methods for handling hot-spots.

7.2 Introduction

One of the most compelling uses of the Internet today is P2P file sharing of multimedia content. Morpheus, the most popular P2P application today with on average 500,000 simultaneous users, provides sharing for MP3 music files (typically in the 3-5 Mbyte range) and video files (today, typically in the 5-100 Mbyte range) [21,48]. This multimedia content is shared among *peers*, which are hosts that intermittently connect to the Internet, typically changing their IP addresses at each new connection. Each peer dedicates a fraction of its disk storage and access bandwidth to the P2P application.

For many university campus networks, P2P content sharing is the dominant traffic type [11]. Campus peers are often independently retrieving the same audio and video content from peers outside the campus, clogging Internet-to-campus pipes and wasting aggregate peer storage [11]. We argue that a university campus can make more efficient use of its resources (WAN bandwidth and peer storage) if the peers in the campus coordinate the replication of their shared content. For example, it is unnecessary that a large fraction of campus users download the same popular MP3s from outside the campus. Instead, the campus peers could maintain a limited number of copies of popular audio/video objects, and the users could access the popular objects from those copies.

The university campus is a special case of what we refer to as a *peer community*. A peer community is a collection of peers that can coordinate the replication of content across the community of peers for the common good of the community. Each peer in a peer community (*i*) shares bandwidth, storage and content with the rest of the peers in the community; (*ii*) has intermittent connectivity, that is, “goes up and down”; (*iii*) works in conjunction with the other peers to dynamically manage the replication of content. We distinguish two types of peer communities: content caches and content clubs.

- In a content cache, a peer first tries to access the desired content from other peers in the community; if the content is unavailable from the community, the community obtains the content on-demand from an outside source (for example, from an external P2P file-sharing system, an external Web server, or from some content repository). The peers in a university or corporate campus can coordinate to form a content cache. Using the storage in its customers’ hosts, a high-speed residential ISP could also create a content cache.

- In a content club, users subscribe to the community to have the right to access any of the audio/video content that the community makes available through its peers. From time to time, the community injects new content into the community of peers. When a peer wants to access an object and the object is unavailable in the community (either because there is no copy of the object in the community or because all the peers with the object are disconnected), then the peer does not have access to the object.

Coordinating the replication of content is central to a peer community. There are several approaches to replicating content. Serverless file systems, such as Farsite [20], provide the strong persistence of a traditional file system by replicating each file the *same* number of times (e.g., 3 times). However, the goals of a file system and a content sharing system are different. File systems strive for strong persistence whereas content sharing systems should strive to satisfy user requests as frequently as possible. Another approach is to allow the content to replicate itself across the peers without coordination, as done today in file-sharing systems such as Napster, Gnutella and Morpheus. Whenever a peer wants to access an object, the peer downloads the object (if available) and puts a copy of the object in its shared storage. However, as mentioned above, this non-coordinated approach leads to excessive replication of popular content, and can also be wasteful of bandwidth.

Schemes for replicating content in a peer community should solve the following difficult problems:

- **Performance:** We want to replicate the content as a function of content popularity in order to maximize the probability that requests will be satisfied within the community (that is, the "hit probability"). We have found the issue of *popularity-dependent replication* to be crucial for a peer community.

- **Adaptive:** We want the content replication to adapt as object popularities change and new objects are introduced into the community.

- **Distributed:** We want the replication mechanism to be distributed, without relying on centralized servers for coordination.

Another important component of a peer community is a means for locating content. Specifically, if a peer X wants an object o, then the peer community should provide X with the IP address of at least one up peer, say Y, which has a copy of o (assuming that some up peer in the community has a copy). Peer X can then directly access o from Y. In the past few years a number of substrates have been designed for locating content [23,28,63,75,83,95] in a community of nodes. This chapter is not concerned with the design of such a substrate; instead, we build replication schemes which sit on top of a content-location substrate.

Figure 7.1 plots the benefits of coordinated replication over non-coordinated replication. In this figure we use a content cache of 100 peers, with each peer being up 50% of the time. Figure 7.1 plots the hit probability as a function of the number of objects that can be placed in each peer's shared storage space. Our focus is on large audio and video objects; thus we are concerned with relatively small peer storage capacities. The figure assumes that requests for the different objects follow a Zipf distribution with parameter .8. Figure 7.1 has two curves. The lower curve is the hit probability for the case when a peer independently retrieves and stores content, without regard to the other peers in the community. It is calculated by summing the probabilities in the Zipf distribution up through the number of objects that can be stored locally. The upper curve is the hit probability that can be achieved with coordinated replication. It is obtained with the distributed replication algorithms described in Section 3. This curve clearly shows the dramatic improvements that can be obtained from coordinated replication, even when each peer is up only 50% of the time.

After reviewing the related work in Section 7.3, we formulate the problem of optimal replication of content over a community of peers in Section 7.4. Taking explicitly into account peer-up probabilities, we formulate an integer programming problem, and indicate how the problem can be efficiently solved by dynamic programming. This optimization theory is admittedly "academic". It assumes *a priori* knowledge of object request probabilities, when indeed these probabilities are changing and new objects are being introduced daily. Nevertheless, the theory enables us to obtain upper bounds on what can be achieved by an adaptive algorithm, and gives insight into how objects should be replicated. In Section 7.5 we consider content caches, that is, communities for which it is possible to retrieve content from outside on-demand when the content is unavailable inside. We propose a series of fully distributive, adaptive algorithms for content replication. We find that

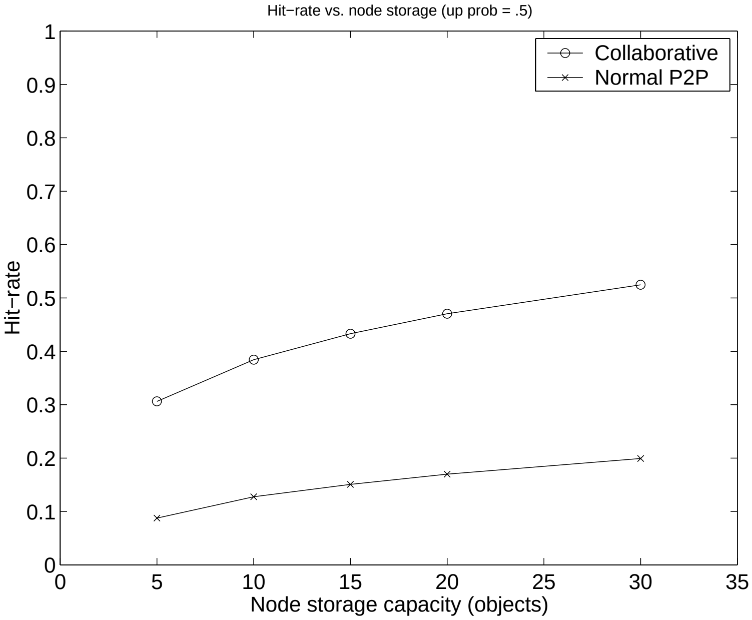


Figure 7.1: Benefits of coordinated replication

the object replacement policy at each of the individual peers plays an important role in performance. In particular, a simple retrieval scheme combined with a distributed form of least-frequently-used (LFU) replacement provides near optimal results. In Section 7.6 we consider content clubs, that is, communities for which all content must be retrieved from inside the community. We simulate a content club for which new content is periodically injected into the community. We find that the number of copies of fresh content injected into the content club plays an important role in content availability. In Section 7.7 we examine the issue of hot spots in peer communities and provide several solutions. Finally Section 7.8 concludes the chapter and outlines future research.

7.3 Related Work

The most prominent P2P file-sharing systems in use today include Morpheus/FastTrack [21, 48], Gnutella [28] Freenet [23] and Napster [51]. These systems provide means to locate and download content. They provide no explicit mechanisms to manage the degree of replication in the set of peers, or for subsets of peer communities (such as a university campus). The notion of content clubs is present in P2P research and development. For example, MojoNation provides incentives to users who provide disk, bandwidth and content resources to the P2P community; Napster now requires its users to pay a membership fee to get access to copyrighted content managed by Napster.

In the research community, there are several ongoing efforts for scalable lookup services. Each of these services can be viewed as a substrate on top of which P2P applications can be built. The substrate provides applications with an API which accepts the name of a content object and returns an up peer that is responsible for that object. Chord [83] uses consisting hashing [37], but relaxes the requirement that all up peers know about all other up peers. A related service is Pastry [75], which forwards messages based on address prefixes rather than on numerical differences with the destination address. See also [63] and [95]. Our work differs from [63,75,83,95] in that our focus is on optimizing availability of content rather than on providing schemes for finding content. Our work layers on top of a content-location substrate.

FarSite [20] is a P2P filesystem with the strong persistence and availability of a traditional filesystem. The Farsite filesystem uses the same number of replicas for each object. The goal of a community is not to provide strong file persistence and availability, but instead maximal content availability. Our goal is similar to that of a video rental store: given limited shelf space, satisfy customer demand by providing a large choice of videos and adequate numbers of copies of popular videos. Thus, in our P2P communities, the number of replicas of an object depends on the popularity and locations of the object.

Squirrel [32] is a decentralized P2P Web cache. It resembles what we call a "content-cache". However, the Squirrel project aims to build a P2P cache that has identical functionality to a centralized Web proxy cache. In particular, it carefully addresses the issue of consistency of cached Web documents. Our focus is on audio and video content, for which consistency is not a primary issue. Furthermore, Squirrel does not address optimal replication of content.

7.4 Optimal Replication of Content over a Community of Peers

In this section we formulate and solve the optimal P2P replication problem. The formulation is somewhat academic as it assumes that the objects and their popularities are known *a priori*. It also assumes that the up probabilities of the peers are known, and the peer-up events are independent. Nevertheless, this formulation and solution provide significant insight into the design of adaptive algorithms for P2P communities (Sections 3 and 4).

Consider a community of I peers. A peer might be a powerful workstation, a personal computer, or an Internet-connected PDA. At any given time, a given peer may be up or down; it may be down because the device is physically disconnected from the communication network or because the peer community application is not launched. For a given peer i, let p_i denote its up probability, that is, the fraction of time that the peer is up. Each peer has private storage and shared storage. Only content in the shared storage can be accessed by the community at large. Denote by S_i the amount of shared storage (in bytes) that the ith peer is prepared to contribute to the community. We suppose that the content in a peer's shared storage is not lost when a peer goes down; when the peer comes back up, all of the content in its shared storage is again available for sharing. (This is generally the case in P2P file-sharing systems such as Napster, Gnutella and Morpheus.)

Let J denote the number of content objects that the community will share. Let b_j denote the size (in bytes) of the jth content object. In this formulation, we assume that the request probabilities for the J objects are known *a priori*. Specifically, we suppose that the request probability for each object j is a known value, q_j, with $q_1 + q_2 + \cdots + q_J = 1$. In this section we define a "hit" to be a request for an object for which a copy of the object is present in one of the up peers in the community. The "hit probability" is the fraction of requests that are hits.

One of the key characteristics of a peer community is that - in order to have a high hit probability within the community - popular objects need to be replicated across peers, with the degree of replication being a function of the object's popularity. Let us denote by n_j the number of replicas of object j; we require each replica to be placed on a different peer. Our goal in this section is to determine $n_1, \ldots, n_J$ so that the hit probability is maximized.

Whenever a peer i wants the content object j, it first searches the community of up peers for a replica of the object. Once finding an up peer i' with the object, peer i will request a direct P2P transfer from peer i'. Peer i may simply stream in and render the object without placing it in its local storage; or it may download the object into the private portion of its local storage. *However, in our design, peer i does not put the object in its shared storage as it is not responsible for sharing the object.*

Having defined the optimal replication problem, we now formulate it as an integer programming problem. To this end, let x_{ij} be a zero-one variable which is equal to one if peer i contains a replica of object j and is zero otherwise. The probability that a given request will have a hit within the community of up peers is

$$P(hit) = \sum_{j=1}^{J} q_j (1 - \prod_{i=1}^{I} p_i^{x_{ij}}) \tag{7.1}$$

Any assignment of replicas to peers must satisfy the local storage constraints:

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i, \quad i = 1, \ldots, I \tag{7.2}$$

The integer programming problem is then to find the 0-1 variables x_{ij} such that (1) is maximized subject to (2). This problem can be shown to be NP-complete by reducing it to the Zero-One Integer Programming problem [26].

The solution to the integer programming problem provides an upper bound on the achievable performance of any distributed, adaptive algorithm (see Sections 3 and 4). Integer programming heuristics for solving this general problem will be explored as part of our future research. For the purposes of this chapter, we solve a special case of this problem, namely, when each node is up with the same probability $p_i = p$. (Our algorithms in the subsequent sections do not require this; but the assumption is useful for benchmarking purposes.) Specifically, consider the problem of maximizing

$$P'(hit) = \sum_{j=1}^{J} q_j (1-p)^{n_j} \tag{7.3}$$

subject to

$$\sum_{j=1}^{J} b_j n_j \leq S \tag{7.4}$$

where $S = S_1 + \cdots + S_I$. Note that, because storage has been aggregated into one constraint, the optimal value for $P'(hit)$ is actually an upper bound on the optimal hit probability (for homogeneous values of p_i); however, in our numerical work, this upper bound turns out to be very tight. This optimization can be solved efficiently by dynamic programming. Indeed, let $f_j(s)$ be the minimum miss probability when there are s bytes of aggregate storage and objects $j, \ldots, J$ to replicate. Then standard dynamic programming arguments give

$$f_j(s) = \min_{n : b_j n \leq s} [f_{j+1}(s - nb_j) + q_j a^n]$$

The optimal replica profile $n_1, \ldots, n_J$ solves $f_1(S)$. As we show in the following section, this optimal replica profile provides insight into how objects should be replicated and it benchmarks the hit probabilities of the adaptive algorithms.

7.5 Adaptive Replication for Content Caches

The main contribution of this chapter is a suite of adaptive, distributed schemes for replicating content on-the-fly in a community of peers. In this section, our focus is on content-cache communities, that is, communities for which requests can be satisfied on-demand from outside the community when the desired object is unavailable from inside the community.

We suppose that each object can be identified by a name. There are a number of current projects and research efforts in the direction of assigning names to objects [8,44] . For example, each object can be identified by a URL or have a unique URN. We also suppose that each peer has a persistent name, which is assigned when the peer initially subscribes to the application.

Our algorithms assume the existence of a substrate with the following functionality. The substrate has a function call that takes as input an object name j and creates internally an ordered list of all the peers (both the up and down peers). Let $i_1, i_2, \ldots$ be the list of peers for o. Let Y_1 be the first up peer on this list, Y_2 be the second up peer on the list, etc. We refer to Y_1 as the first-place winner for j among the current set of up peers; more generally, we refer to Y_k as the kth place winner. The function call returns $Y_1, Y_2, \ldots, Y_K$

for any desired value of K. The peer Y_1 is also referred to as the "home" of j, i.e., it is the best place for j among the current set of up peers. There are number of substrates today that provide homes, including Chord [83], Pastry [75] and CAN [63]. These substrates are different in their overlay topology, message passing complexity, and message processing complexity, but they all scale and are fully distributed. These substrates can be extended to provide kth-place winners. Thus, any of these existing substrates can be used by our replication algorithms.

A simple example of such a substrate is what we call *two-argument hashing*. This substrate, inspired from CARP [90] uses a hash function that is a function of both (i) the object names, and (ii) the peer names. Specifically, let $h(i,j)$ be a hash function that maps the object name j and the peer name i into the hash space $[0,1]$. For example, $g(\cdot)$ could be a hash function that maps arbitrary ASCII strings into $[0,1]$, and $h(i,j)$ could be $h(i,j) = g(i+j)$, where $i+j$ is the concatenation of i and j (assuming that both object names and peer names are ASCII strings). We require that the mapping be uniformly distributed over $[0,1]$ and that collisions are rare. With two-argument hashing, the substrate applies the function $h(i,j)$ to each up peer, and the K up peers with the highest values become the top-K winners, $Y_1, \ldots, Y_K$. One of the nice features of two-argument hashing is that it uniformly distributes objects to the various homes. Our replication algorithms do not require the two-argument hashing substrate; we present it here to serve as a simple illustrative example of how the top-K winners may be calculated. The substrates in [83] and [75] use single-argument hashing; they hash both the object name j and the peer name i separately to determine the home peer.

We can now present our first adaptive algorithm for replication. Suppose X is a peer that wants object j. X will get access to j as follows:

Adaptive Top-K Algorithm

1. X uses the substrate to determine Y_1, the current first-place winner for j.

2. X asks Y_1 for j.

   - If Y_1 doesn't have j, Y_1 determines $Y_2, \ldots, Y_K$ and pings each of the $K-1$ peers to see if any of them have j.

   - If any of $Y_2, Y_3, \ldots, Y_K$ have j, Y_1 retrieves j from one of them and puts a copy in its shared storage. If none of them have j (a "miss" event), Y_1 retrieves j from outside the community and puts a copy in its shared storage.

   - If Y_1 needs to evict an object to make room for j in its shared storage, Y_1 uses the LRU (least recently used) replacement policy.

3. Y_1 makes j available to X (either for streaming or for downloading into X's private storage). Note that X does not put j in its shared storage unless $X = Y_1$.

The idea behind the algorithm is as follows. Each object j has attractor nodes, namely, the leading (up and down) nodes $i_1, i_2, \ldots$ determined by the underlying substrate. The object j tends to get replicated in its attractor nodes, which go up and down over time.

Queries for objects also tend to get sent to attractor nodes. Thus, a query for a particular object j tends to get directed to up nodes that likely have the object. Popular objects get replicated on-the-fly when none of the top-K peers have the requested object-the-fly when none of the top-K peers have the requested object.. For the case $K = 1$, only the first-place peer is checked for the object before going to the outside. By making $K > 1$, the community checks a larger number of candidates before resorting to the outside.

We see that the Adaptive Top-K Algorithm possesses many desirable properties. It replicates content on-the-fly without any *a priori* knowledge of object request patterns. It is fully distributed. It is possible that there will be a miss even when the desired object is in some up peer in the community; however, we shall show that if K is appropriately chosen, the probability of such a miss is low.

There are three key components to the Top-K algorithm:

1. **Management of private and shared storage.** A peer does not store a copy of an object in its shared storage unless it has obtained it on the behalf of another peer.

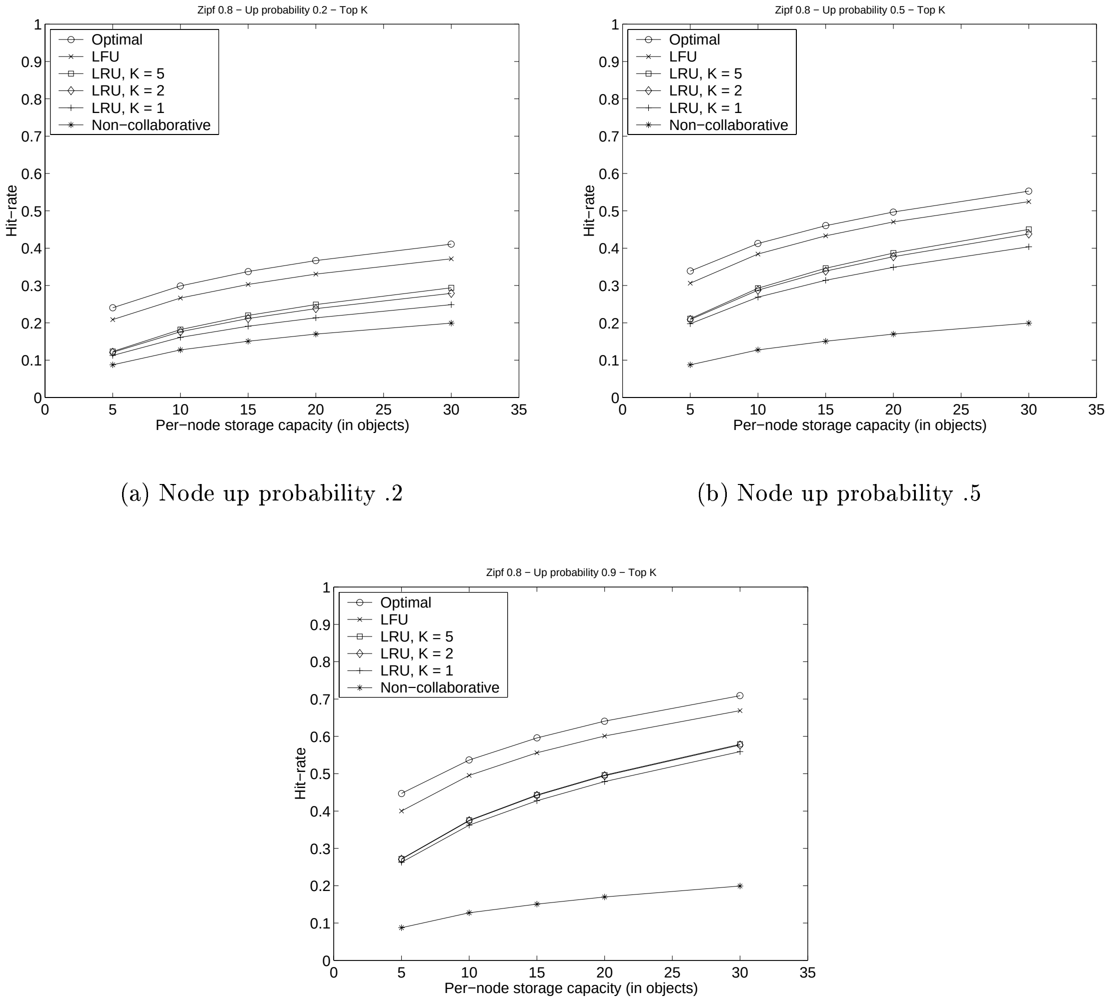
2. **Replacement policy.** When a peer introduces a copy of an object in its shared storage, it may have to evict one or more other objects.

3. **On-demand replication and search.** If the current first-place winner does not have the requested object, it acquires and replicates the object. Note that other replicas of the same object may be present in some kth-place winner or in some down peer.

To study the hit probability performance, we have performed simulation experiments with 100 peers and 10,000 objects. Object sizes are randomly distributed with sizes uniformly distributed between b and $2b$ for a fixed value of b. For each experiment, we have assumed that each peer contributes the same amount of shared storage to the community; our experiments run from 5 objects per peer on average to 30 objects per peer on average. For simplicity, in each experiment all the peers have the same up probability. We have considered three up probabilities: .2, .5 and .9. We suppose that the request probabilities for the various objects follow the Zipf distribution; our experiments use Zipf parameters .8 and 1.2 [82]. For the underlying substrate, for convenience we have used the two-argument hash function. (The underlying substrate can have an impact on performance, as they do not distribute objects to homes in the same uniform manner.)

Figures 7.2 and 7.3 show six graphs corresponding to the six different combinations of up probabilities and Zipf parameters. Each graph plots hit probabilities as a function of peer storage. The top curve in each of these figures is the theoretical optimal, as described in Section 2. Each figure has a curve for $K = 1$, $K = 2$ and $K = 5$. (In all of our experiments, no significant improvement was obtained by increasing K beyond 5.) The bottom curve is the hit probability for the case when each of the peers independently retrieves and stores content, without regard to the other peers in the community; we refer to this as the "selfish algorithm". We make the following observations:

- As we would intuitively expect, the hit probability increases if we increase the peer storage capacity, the peer up probability, or the Zipf parameter.



(a) Node up probability .2

(b) Node up probability .5

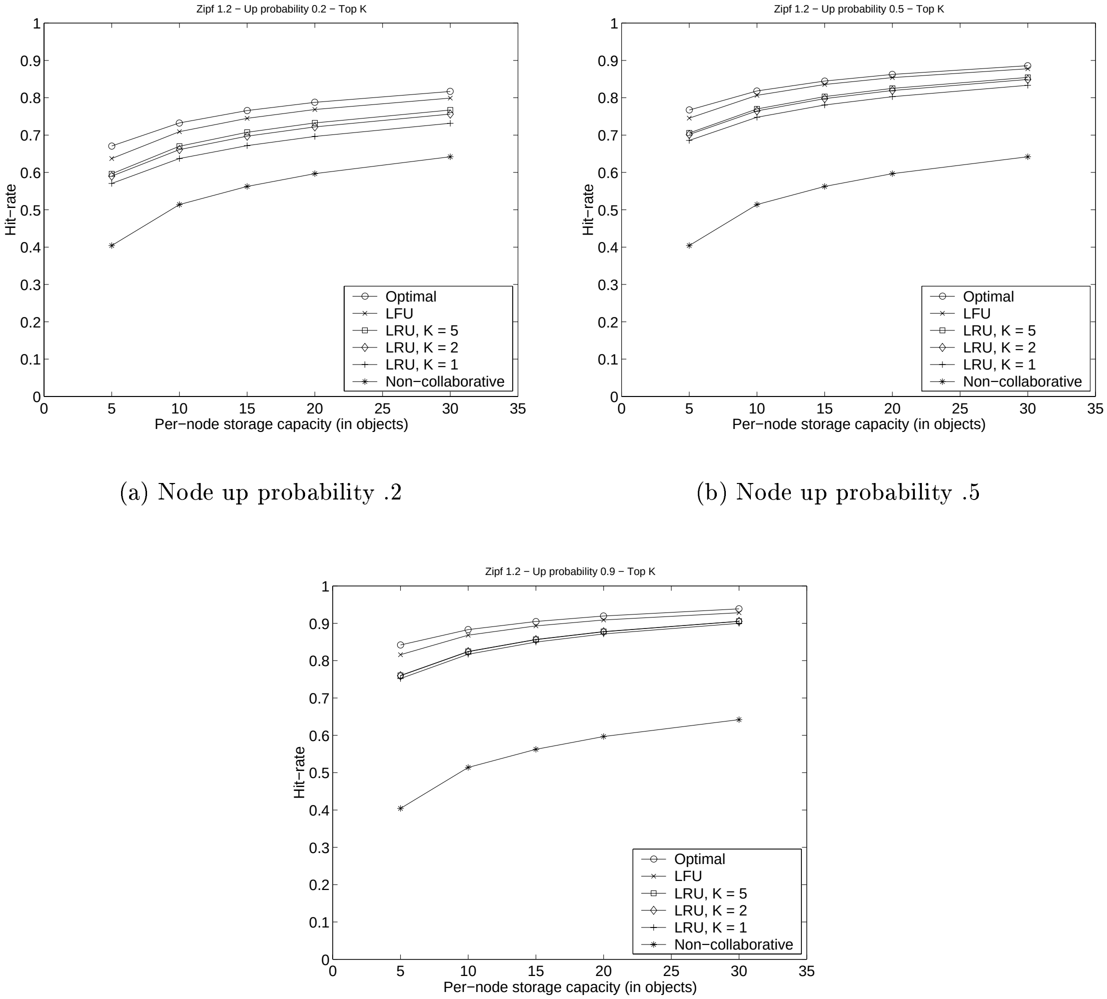
(c) Node up probability .9

Figure 7.2: Hit-rate as function of node storage capacity for Zipf parameter .8

- The adaptive algorithm with $K = 1$ performs significantly better than the selfish algorithm, but significantly worse than the theoretical optimal.

- Using a K value greater than 1 improves the hit probability, especially when peers are frequently down. $K = 2$ provides an important improvement; further increasing K only gives a marginal improvement. Figure 7.4 shows the fraction of misses for which the object was indeed available in some up peer for Zipf =.8.

Examining how objects are replicated provides important insight. Figures 7.5 and 7.6 show, as a function of object popularity, the number of replicas per object for the theoretical optimal and for the adaptive algorithm with $K = 1$. For the adaptive algorithm, the number of replicas per object is changing over time; the graphs report the average



(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.3: Hit-rate as function of node storage capacity for Zipf parameter 1.2

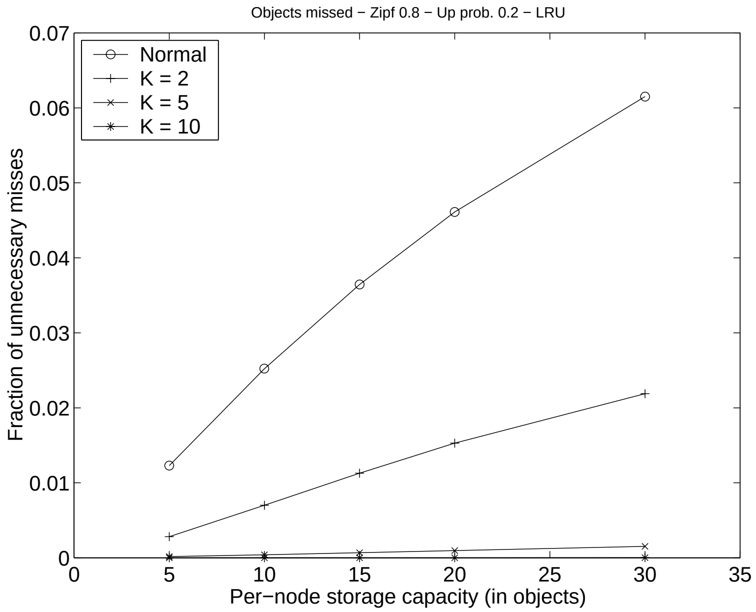


Figure 7.4: Fraction of misses for Zipf = .8

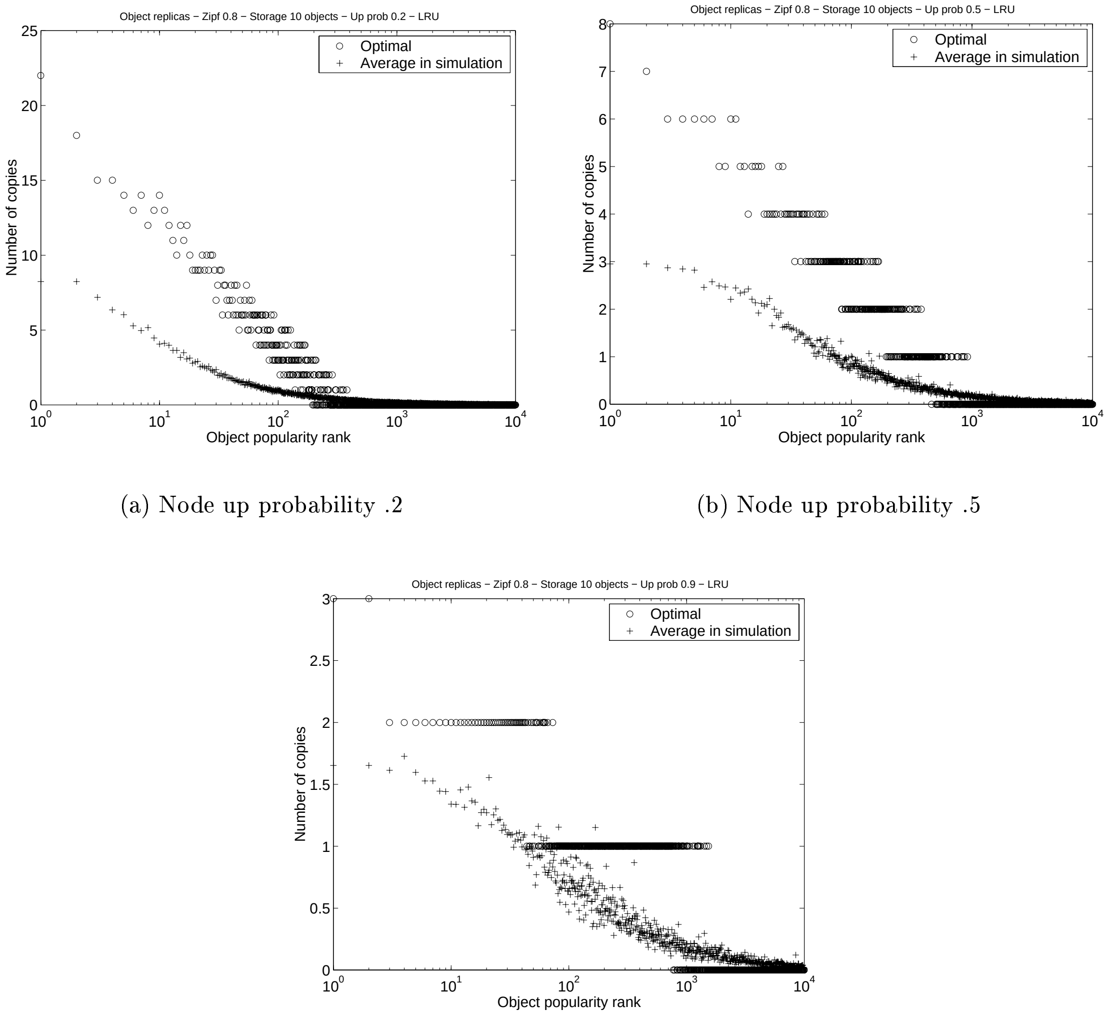
values. Again, the figure presents six graphs, one for each pair of up probability and Zipf parameter. The difference in how the theoretical optimal and the adaptive algorithm replicate objects is striking. The optimal algorithm replicates the more popular objects much more aggressively than does the adaptive algorithm; and it doesn't store the less popular objects, whereas the adaptive algorithm provides temporary caching to the less popular objects.

7.5.1 Distributed LFU Replacement

The reason why the adaptive algorithm provides temporary caching to the less popular objects is that whenever one of these objects is requested, it gets cached in the community of peers and lingers in the community until it is evicted with the LRU replacement policy. Intuitively, if we stop caching the less popular objects, the popular objects will grab the vacated space and there will be more replicas of the popular objects.

To this end, we introduce the *distributed least-frequently-used (LFU)* replacement policy. Here each peer, for example Y, keeps track of all the requests that have been made to it. Specifically, for every object for which Y has seen a request as a first-place winner, it keeps a count of the number of requests for that object. (In practice, the request counts would be dampened with a weighted exponential moving average. LFU would also stop tracking objects that have very small count values.) It keeps this request count for all objects it has seen requests for, not just for the objects currently stored in its shared storage.

Now suppose that X asks Y_1 for the object j. As in the Adaptive Top-K Algorithm, if Y_1 doesn't have j, it obtains j (either from outside the community or from one of the other K winners) on the behalf of X. Y_1 delivers j to X, but will only put j in its shared

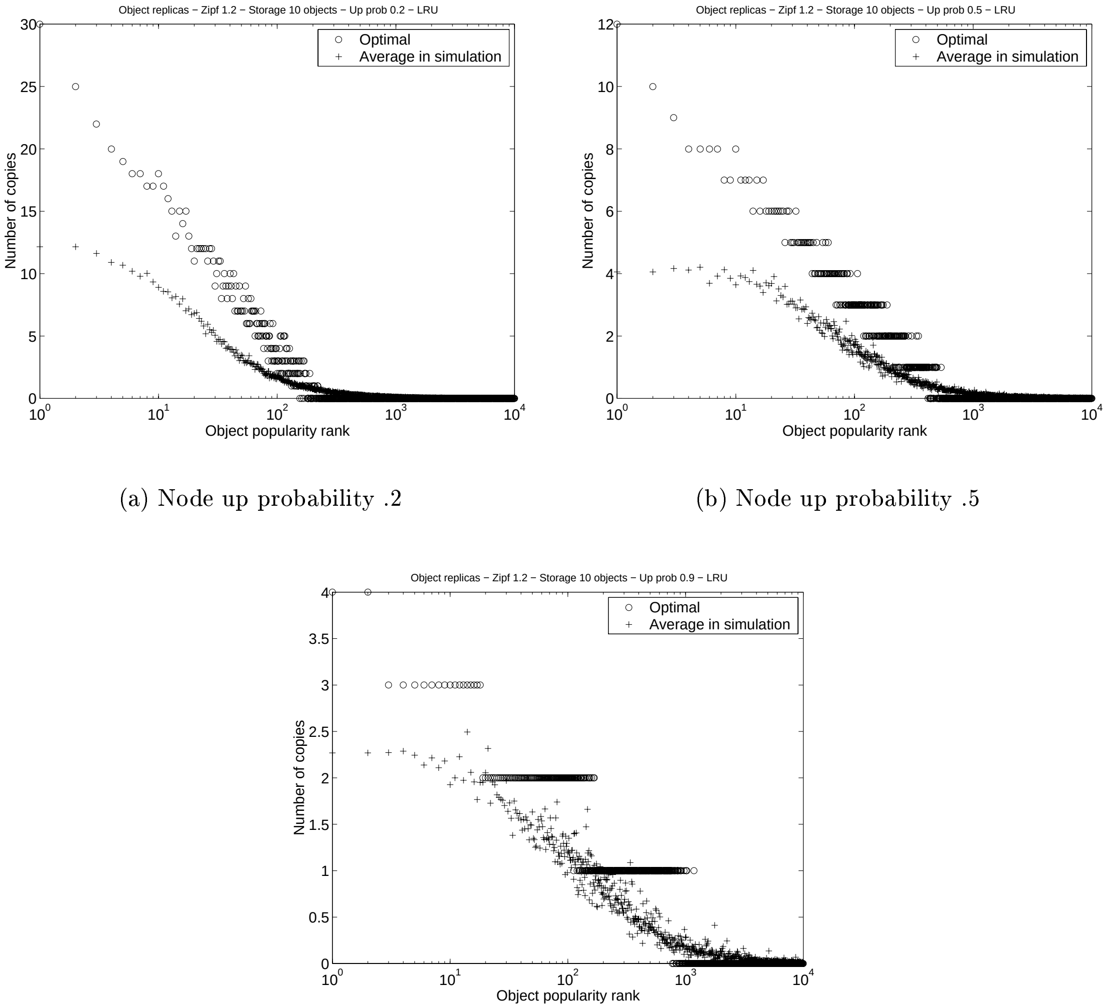


(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.5: Number of replicas per object with 10 objects of per-node storage capacity and LRU replacement policy for Zipf parameter .8



(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.6: Number of replicas per object with 10 objects of per-node storage capacity and LRU replacement policy for Zipf parameter 1.2

storage if its local request count is greater than *the smallest request count* of all the other objects currently cached in Y_1. In this manner, Y_1 does not cache rarely requested objects.

The hit probabilities for distributed LFU with $K = 1$ is also shown in Figures 7.2 and 7.3. We see that the addition of distributed LFU greatly improves the hit probabilities, bringing them in all cases close to the theoretical optimal. Our numerical results (not reported) also showed that $K > 1$ did not improve performance when distributed LFU is used. Figures 7.7 and 7.8 show that the the average number of replicas per object is very close to those in the theoretical optimal. Thus distributed LFU greatly improves performance and simplifies network messaging since $K = 1$ is sufficient. On the other hand, it requires that each peer locally track object request counts, even for objects that are not currently in its local storage.

Approximate Performance Analysis

We now provide an analytical approximation of the hit probability for the Top-K Algorithm with distributed LFU. This analytical technique not only provides an accurate approximation of the hit probability but also sheds considerable insight on the mechanics of the algorithm. To this end, let λ be the rate at which the peer community requests objects, and let $\lambda_j = \lambda q_j$ be the request rate for object j. Let $i_k(j)$ be the kth-place winner for object-j among all the up and down (all I) peers.

Let $\gamma_j(i)$ denote the approximate rate at which requests for object j arrive at peer i when peer i is up. To calculate $\gamma_j(i)$, denote by k the "place" of object j at node i, that is, for fixed i and j, k is such that $i_k(j) = i$. If $k = 1$, then $\gamma_j = \lambda_j$; if $k = 2$, then $\gamma_j(i) = \lambda_j(1 - p_{i_1}(j))$; more generally,

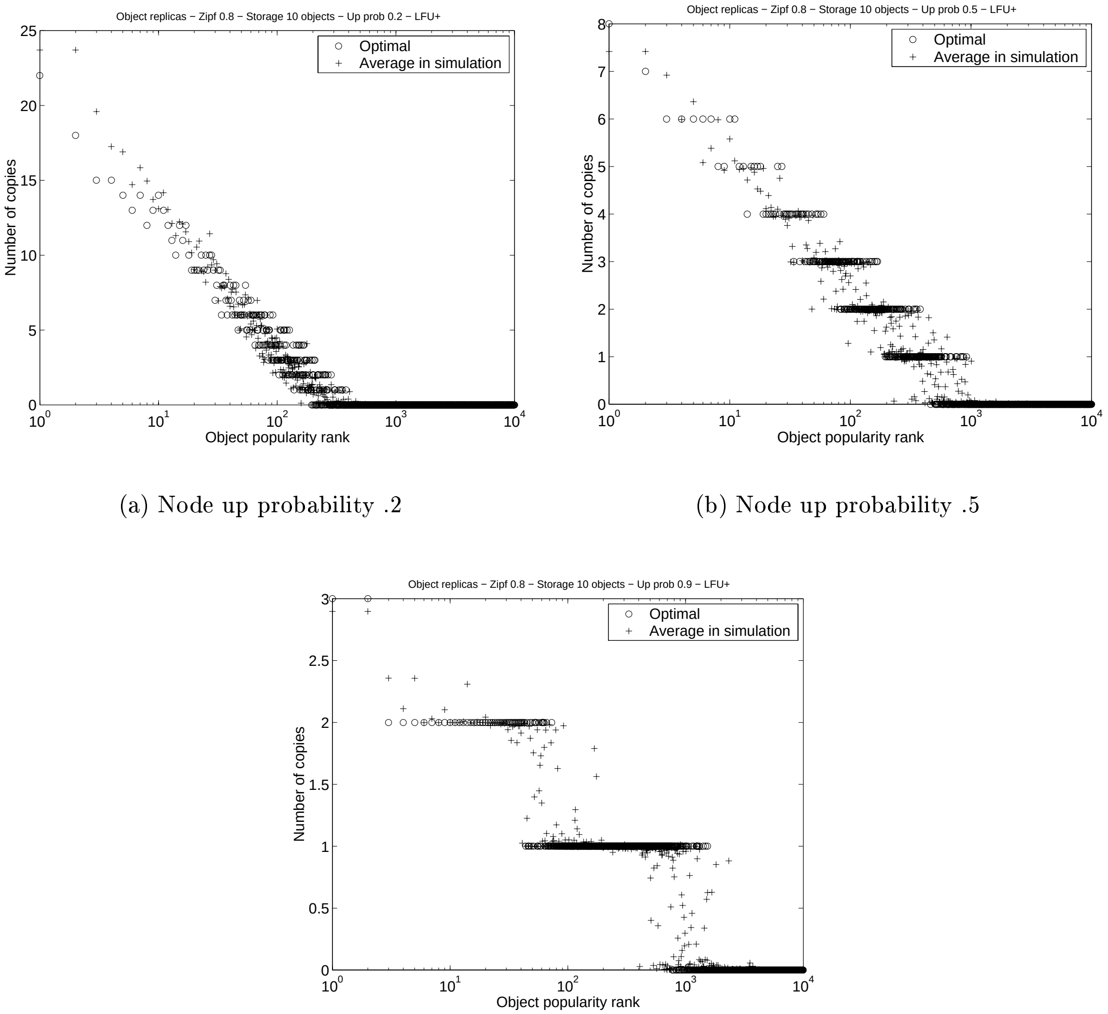
$$\gamma_j(i) = \lambda_j(1 - p_{i_1(j)})(1 - p_{i_2(j)}) \cdots (1 - p_{i_{k-1}(j)})$$

For LFU, the objects for which there are the most requests to peer i are stored in peer i. Re-order the objects so that $\gamma_j(i)$, $j = 1, \ldots, J$, goes from highest to lowest; let J_i be the largest J' such that

$$\sum_{j=1}^{J'} b_j \leq S_i.$$

Also, let $\mathcal{O}_i$ be the set of first J_i objects in the re-ordered list of objects. The set $\mathcal{O}_i$ is the approximate set of objects that are stored in peer i. The hit probability for a request for object j is approximately the probability that at least one peer i with $j \in \mathcal{O}_i$ is up. Thus

$$P(hit) = \sum_{j=1}^{J} q_j[1 - \prod_{i:j \in \mathcal{O}_i} (1 - p_i)]$$

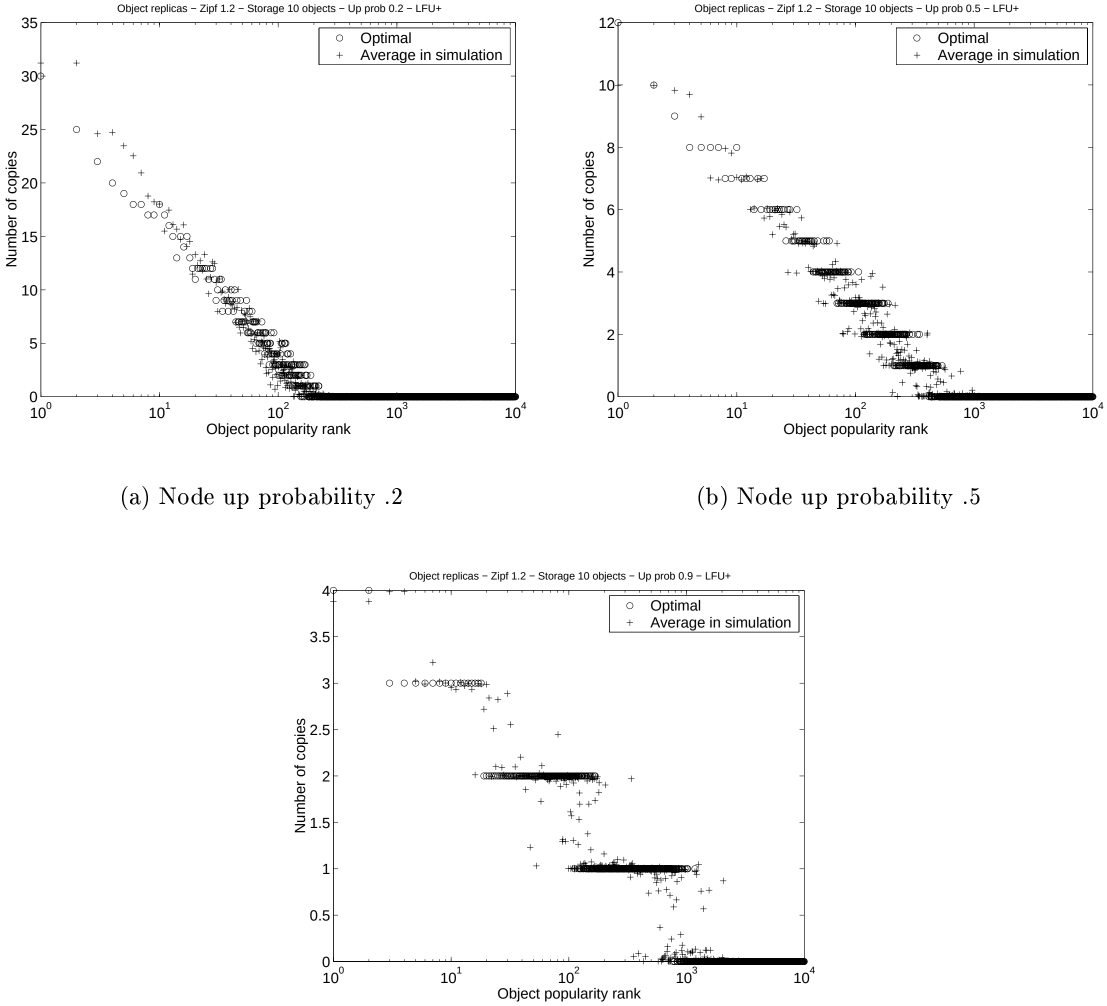


(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.7: Number of replicas per object with 10 objects of per-node storage capacity and LFU replacement policy for Zipf parameter .8

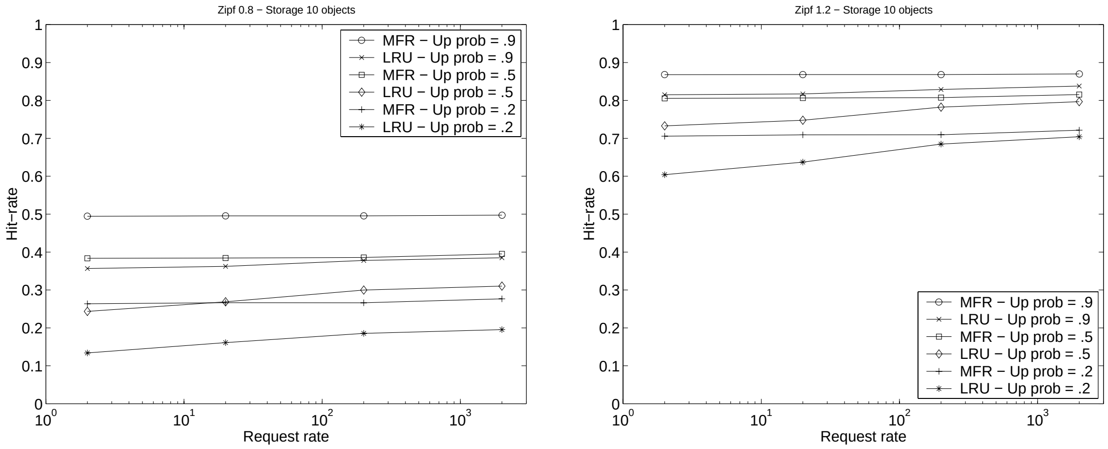


(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.8: Number of replicas per object with 10 objects of per-node storage capacity and LFU replacement policy for Zipf parameter 1.2



(a) Zipf .8

(b) Zipf 1.2

Figure 7.9: Sensitivity of hit probabilities to request rate

7.5.2 Timescales

There are two important timescales in peer communities. At the slower timescale, peers in the community are going up and down (for example, connecting and disconnecting their personal computers). At the faster time scale, up peers are requesting objects. In our numerical work, we set the rate at which each peer goes down to 1, and the per-peer request rate to 20. (Thus, on average, each peer makes 20 requests before going down.) The up rates were set to .2, 1 and 9 to obtain the three different up probabilities. Figure 7.9 explores the sensitivity of the hit probabilities of different algorithms for a wider range of request rates, namely, request rates of 2, 20, 200, and 2000. We see from Figure 7.9 that hit rates are largely insensitive to the request rates.

7.6 Adaptive Replication for Content Clubs

In this section we consider adaptively replicating objects in content clubs. The key difference between a content club and a content cache is that in a content club all content is only available from members of the club. If a request cannot be satisfied from one of the club members, the object cannot be delivered. This scenario is similar to a video rental store, where the store has a certain number of copies for a given video and if all copies are rented, then a new customer cannot get that video.

The club makes new content available by injecting multiple copies into the community of peers. When new content is introduced, some peers may have to remove existing unpopular objects from their storage to make room for the new content.

Our replication scheme for content clubs is as follows. When we insert a new object, we create L copies of it, by inserting it in the peers $Y_1, \ldots Y_L$, i.e., the top L winners among the current up peers. Note that $Y_1, \ldots Y_L$ may not be the true top L winners among all peers because some of the true winners may be down. Each peer also runs the LRU replacement algorithm on its storage and new objects evict older objects. When a peer wants to request an object, it uses the Adaptive Top-K Algorithm from Section 7.5. However, if the first-place winner Y_1 is unable to find the object from the top-K winners, the request cannot be satisfied and we incur a miss.

We simulate our replication scheme as follows. The club starts initially with a set of J objects which are replicated on the peers according to the algorithm above. We call the current set of J objects as the *active set of objects*. Periodically we replace 10% of the objects in the active set with new objects. We remove from the request distribution the most unpopular objects and insert the new objects and randomly determine their popularity rank. The request probabilities are recalculated after the objects have been introduced to make them correspond to the Zipf-distribution. Peers can request only objects in the active set, although previously active objects may linger on in the peer storage. Each peer runs the LRU replacement algorithm and objects that have been removed from the active set will eventually get removed from all the peers in the community.

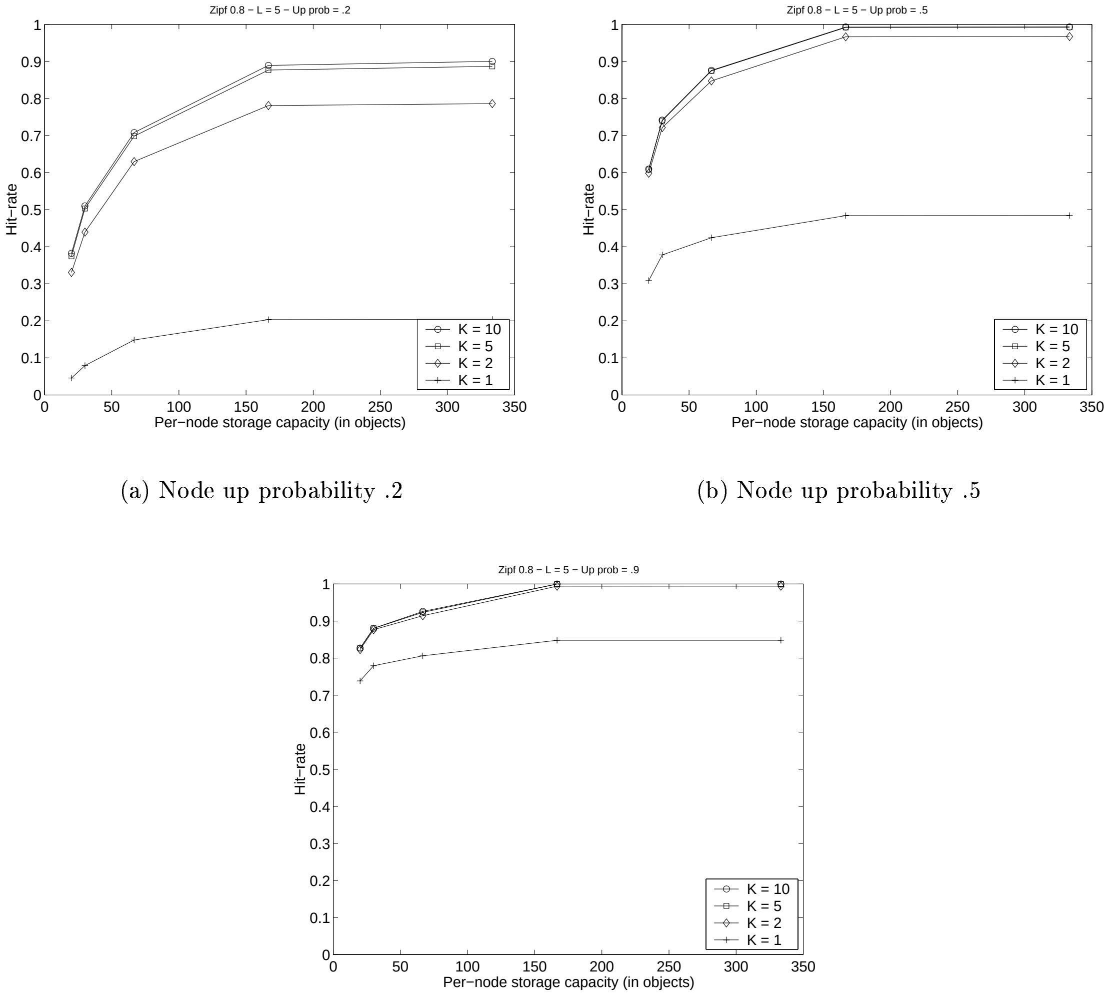
In Figures 7.10 and 7.11 we show the hit-rates for a community of 100 peers, Zipf parameter .8, and two values of L, 5 and 10. Results for Zipf parameter 1.2 were similar and are not shown. We can see that the performance depends quite significantly on the value of L. For small peer sizes, a large value of L decreases performance. This is because every time a new object is introduced, we create L copies of it in the community. If the peers have only a small amount of storage, inserting a large number of copies of an object can force out of the community copies of more popular objects, thereby reducing the overall hit probability. If the peers have more storage, then the evicted objects tend to be less popular.

Figures 7.12 and 7.13 show the hit-rate as a function of L and K for 2 different node storage capacities for Zipf parameter .8 and node up probability .5. We make the following observations:

- There is a dramatic improvement in the hit probability when increasing K from 1 to 2. Indeed, if $K = 1$, only the peers that originally acquired a particular object (during original object insertion) ever hold the object. If a first-place winner comes up after insertion, it will receive queries for the object but never obtain it. Further increases in K have only marginal improvement unless up probabilities are very small.

- There is some value of L which yields the best performance in the given situation. This value depends on the overall storage capacity and peer up probabilities. The smaller the storage capacity, the smaller the optimal value of L because we need to be more careful how to use the limited storage capacities.

We have assumed in the simulation that L copies of each new object are inserted into the community. For many applications in practice, there will be some *a priori* knowledge of the relative popularity of new objects; this knowledge can be used to set L to different values for different new objects.

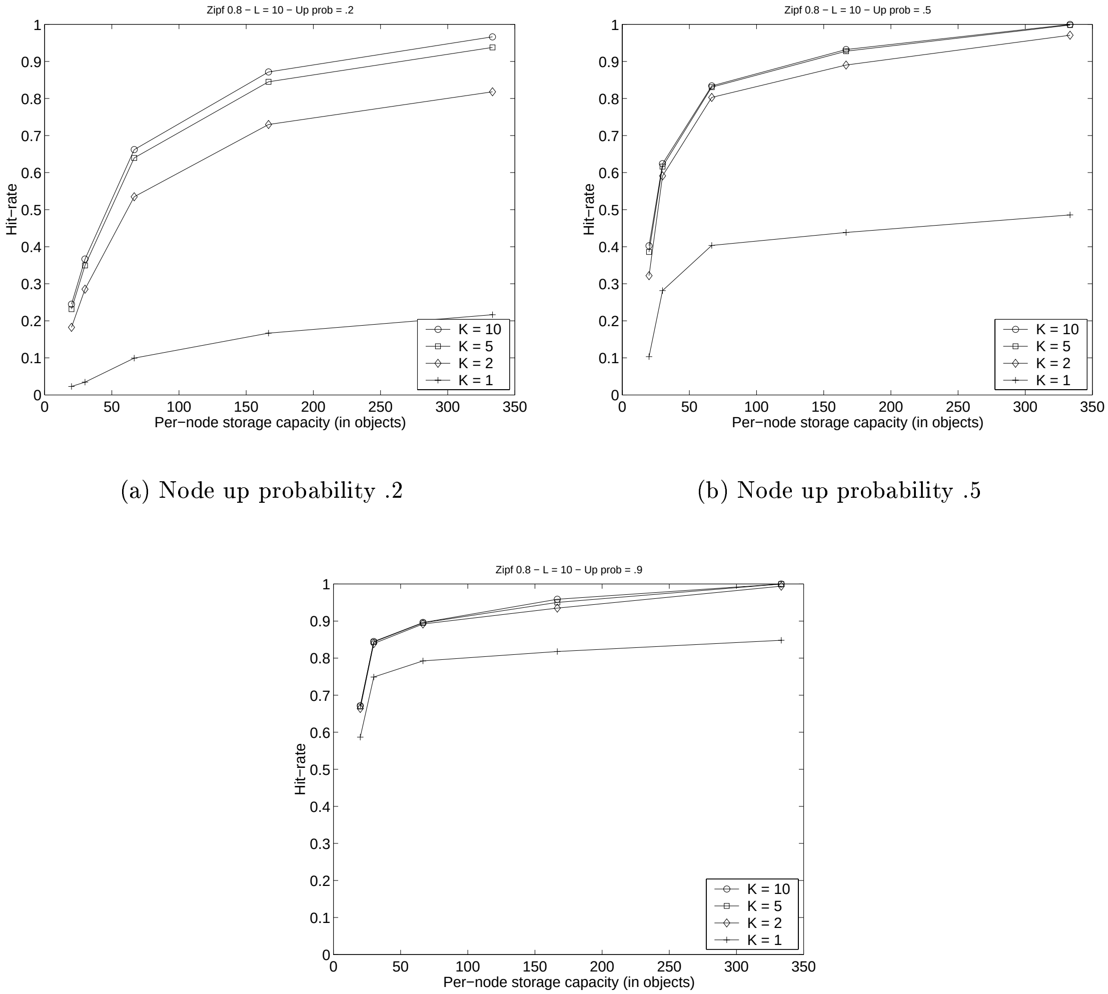


(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.10: Hit-rate as function of node storage capacity for Zipf = .8 and $L = 5$

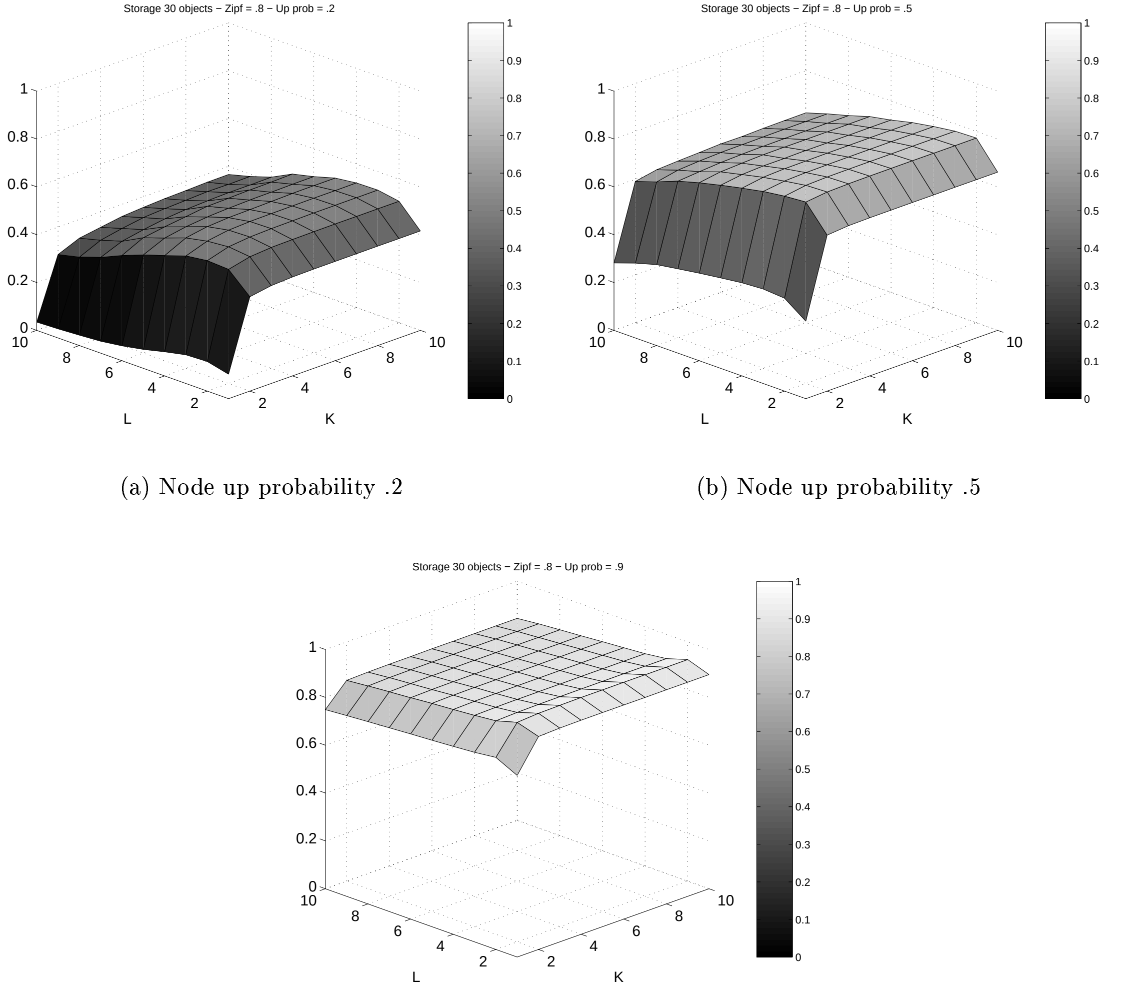


(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.11: Hit-rate as function of node storage capacity for Zipf = .8 and $L = 10$



(a) Node up probability .2

(b) Node up probability .5

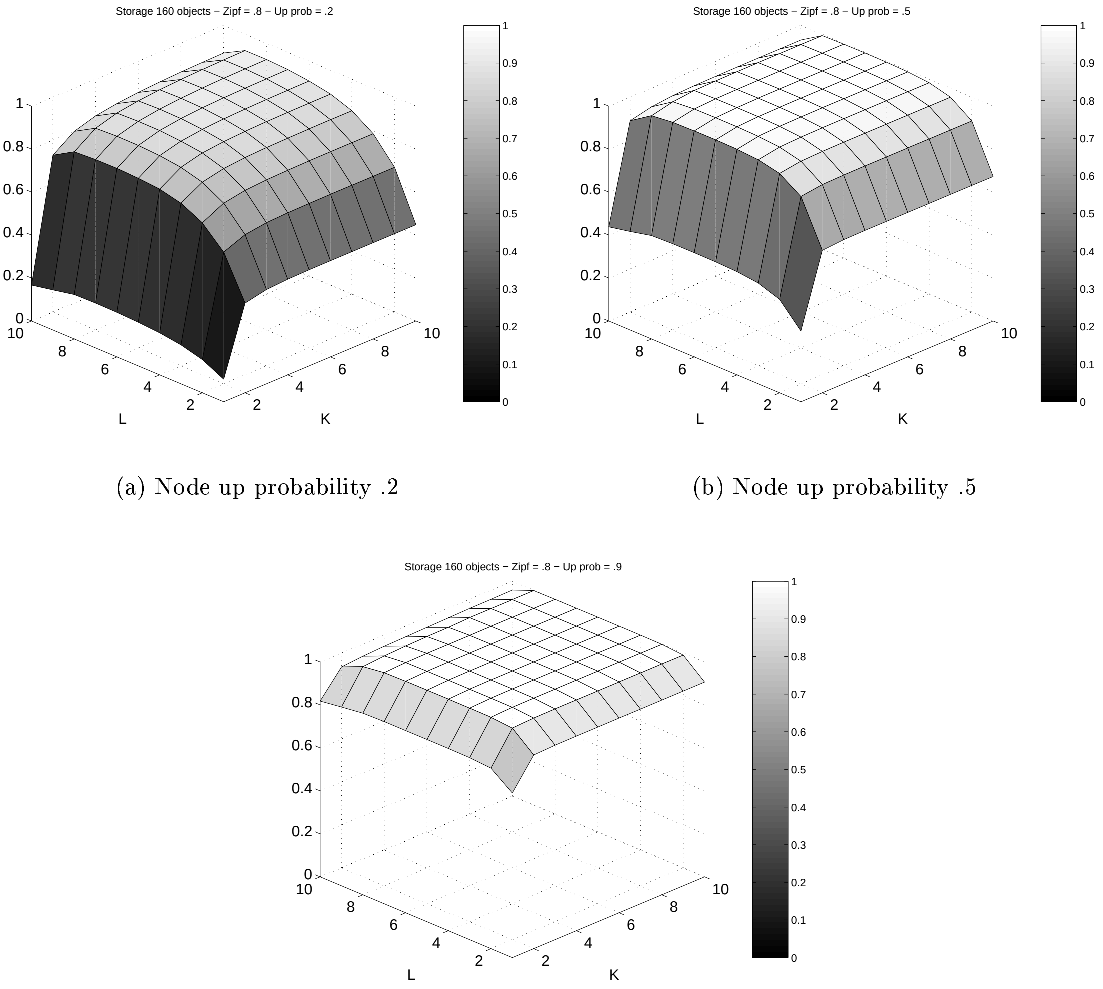
(c) Node up probability .9

Figure 7.12: Hit-rate as function of L and K for Zipf $=$ .8 and 30 objects of per-node storage

For content clubs we use LRU replacement rather than distributed LFU. Indeed, LFU is not as natural in content clubs, since when a a new object is introduced to the community, its request count is zero, and is therefore not stored anywhere. We have experimented with a number of variations of distributed LFU for content clubs and have seen no significant improvement.

7.7 Dealing with Hot Spots

The replication theory and algorithms in the previous sections maximize the availability of requested objects. However, they do not directly address the issue of *hot spots*. For



(a) Node up probability .2

(b) Node up probability .5

(c) Node up probability .9

Figure 7.13: Hit-rate as function of L and K for Zipf = .8 and 160 objects of per-node storage

example, suppose that j is among the most popular objects, and that Y_1, the first-place winner for j, is always up. Then the algorithms of the previous sections will create exactly one replica for o and store it in Y_1. If the request rate for j is very high, then Y_1 will become overloaded.

Optimization Theory

In this section we modify the theory and algorithms of the previous sections to better respond to hot spots. First consider the optimization theory of Section 7.4. Suppose now that each peer i announces a value α_i, which is the maximum average rate at which it is willing to serve content to the rest of the community. Also let λ be the aggregate request rate for objects.

Suppose a copy of object j is in peer i. Let us calculate the transmission rate of peer i due to object j. We suppose that the aggregate transmission rate (across all peers) for object j is equally shared among all the up peers that have a copy of j. The average number of up peers with a copy of j is $\sum_k p_k x_{kj}$. Thus the transmission rate for peer i due to type-j objects is

$$\lambda q_j b_j \frac{1}{\sum_{k=1}^{I} p_k x_{kj}}.$$

Summing up over all objects j gives the following constraint:

$$\sum_{j=1}^{J} \lambda q_j b_j \frac{x_{ij}}{\sum_{k=1}^{I} p_k x_{kj}} \leq \alpha_i \qquad (7.5)$$

Therefore, the integer programming formulation of the optimal replication problem with explicit load balancing is to maximize (7.1) subject to (7.2) and (7.5). We will address this problem in our future research.

Now consider the case when all the p_i's are equal. A natural constraint is that the average transmission load due to any object j placed on any peer be less than some fixed value, say, β', that is,

$$\frac{\lambda q_j b_j}{p n_j} \leq \beta'$$

or equivalently

$$n_j \geq \beta q_j b_j \qquad (7.6)$$

where $\beta = \beta' \lambda / p$. This constraint can easily be included in the dynamic programming formulation. (However, we recommend rounding the right-hand side, thereby allowing for zero copies of less popular objects.)

Adaptive Algorithms

To deal with hot spots, we should abide to the following two principles:

- **Replicate:** When the peers holding a hot object are overloaded, the peers should push the object to the "next-place winners". For example, if the first- and second-place winners are overloaded with requests for object o, they should push a copy of o to third (and possibly fourth, fifth, etc.) place winners.

- **Spread Requests:** When a peer X wants an object o, it should ping the top-K winners. If more than one of the K peers has o, it should choose one at random and request the object from that peer. In the case of *content caches*, if no peer has o, X should request o from the first-place winner Y_1 (which will request o from outside the community on the behalf of X.

We propose the following algorithm for replication. When a peer Y sees that it is overloaded with transmissions for an object o, it determines the the current top-R winners and asks these winners if they have o. Let $i_1, i_2, \ldots, i_r$ be the peers that indicate that they do not have o, ordered by their winner placement for object o. Y will then push o into $i_1, \ldots, i_s$ $(s \leq r)$, with the value of s depending on the extent of overload and the number of top-K peers that currently have o.

Chopping Objects into Small Pieces

Given the focus of this chapter is on large multimedia objects, another approach to load balancing is to chop up each object into small pieces, and give each piece a unique name. This will cause each new object introduced into the community to be scattered among the peers of the community. When a peer accesses an object, the transmission load will be shared by a large number of peers.

One drawback of this approach is that if only one piece is unavailable from the peer community, then there will be miss. Such a miss event is particularly unfortunate for a content club, since the missing piece cannot be retrieved from the outside. Therefore, it is useful to introduce redundant pieces so that, say, any n of $n + m$ pieces are needed to reconstruct the object. This will be the subject of future research.

7.8 Conclusion

In this chapter we have studied content replication in peer-to-peer communities. We distinguish between two types of coordinated peer communities: content caches, for which content can be retrieved from outside the community if unavailable from within the community; and peer clubs, for which all content must be retrieved from inside the community. We have formulated an integer programming problem that provides us with the exact optimal solution for both peer caches and content clubs. We have developed several adaptive, distributed algorithms for replicating content on-the-fly. Through extensive experiments, we have found that our algorithms combined with a distributed least-frequently-used replacement policy provide near-optimal performance, both in terms of hit-rate and number of replicas.

The research in this chapter provides for many opportunities for future research. We plan to explore integer programming heuristics to better estimate the optimal solution in equation (7.1) and also study the load-balanced version of the original problem. We will also study if chopping the large objects into small pieces can help in improving the performance.

Chapter 8

Distribution of Layered Encoded Video

8.1 Overview

The efficient distribution of stored information has become a major concern in the Internet which has increasingly become a vehicle for the transport of stored video. Because of the highly heterogeneous access to the Internet, researchers and engineers have argued for layered encoded video. In this chapter we investigate delivering layered encoded video using caches. Based on a stochastic knapsack model we develop a model for the layered video caching problem. We propose heuristics to determine which videos and which layers in the videos should be cached. We evaluate the performance of our heuristics through extensive numerical experiments. We also consider two intuitive extensions to the initial problem.

8.2 Introduction

In recent years, the efficient distribution of stored information has become a major concern in the Internet. In the late 1990s numerous companies – including Cisco, Microsoft, Netscape, Inktomi, and Network Appliance – began to sell Web caching products, enabling ISPs to deliver Web documents faster and to reduce the amount of traffic sent to and from other ISPs. More recently the Internet has witnessed the emergence of content distribution network companies, such as Akamai and Sandpiper, which work directly with content providers to cache and replicate the providers' content close to the end users. In parallel to all of this caching and content distribution activity, the Internet has increasingly become a vehicle for the transport of stored video. Many of the Web caching and content distribution companies have recently announced new products for the efficient distribution of stored video.

Access to the Internet is, of course, highly heterogeneous, and includes 28K modem connections, 64K ISDN connections, shared-bandwidth cable modem connections, xDSL connections with downstream rates in 100K-6M range, and high-speed switched Ethernet connections at 10 Mbps. Researchers and engineers have therefore argued that layered encoded video is appropriate for the Internet. When a video is layered encoded, the number of layers that are sent to the end user is a function of the user's downstream bandwidth.

An important research issue is how to efficiently distribute stored layered video from servers (including Web servers) to end users. As with Web content, it clearly makes sense to insert intermediate caches between the servers and clients. This will allow users to access much of the stored video content from nearby servers, rather than accessing the video from a potentially distant server. Given the presence of a caching and/or content distribution network infrastructure, and of layered video in origin servers, a fundamental problem is to determine *which videos* and *which layers in the videos* should be cached. Intuitively, we will want to cache the more popular videos, and will want to give preference to the lower base layers rather than to the higher enhancement layers.

In this chapter we present a methodology for selecting which videos and which layers should be stored at a finite-capacity cache. The methodology could be used, for example, by a cable or ADSL access company with a cache at the root of the distribution tree. Specifically, we suppose that the users have high-speed access to the cache, but the cache has limited storage capacity and a limited bandwidth connection to the Internet at large. For example, the ISP might have a terabyte cache with a 45 Mbps connection to its parent ISP. Thus, the video caching problem has two constrained resources, the cache size and the transmission rate of the access link between the ISP and its parent ISP. Our methodology is based on a stochastic knapsack model of the 2-resource problem. We suppose that the cache operator has a good estimate of the popularities of the the video layers. The problem, in essence, is to determine which videos and which layers within the video should be cached so that customer demand can best be met.

This chapter is organized as follows. In Section 8.3 we present our layered video streaming model. In Section 8.4 we present our utility heuristics and evaluate their performance. Section 8.5 extends our caching model by adding the possibility to negotiate the delivered stream quality. Section 8.6 considers a queueing scheme for managing client requests. Section 8.7 considers the usefulness of partial caching. Section 8.8 presents an overview of related work and Section 8.9 concludes the chapter.

8.3 Model of Layered Video Streaming with Proxy

Figure 8.1 illustrates our architecture for continuous media streaming with proxy servers. We first give a rough overview of our streaming architecture and then discuss each component in detail. All available continuous media objects are stored on the origin servers. Popular streams are cached in proxy servers. The clients direct their streaming requests to the appropriate proxy server. If the requested stream is cached in the proxy, it is directly streamed over the local access network to the client. If the requested stream is not cached in the proxy, it is streamed from the origin server over the wide area network to the proxy. The proxy forwards the stream to the client.

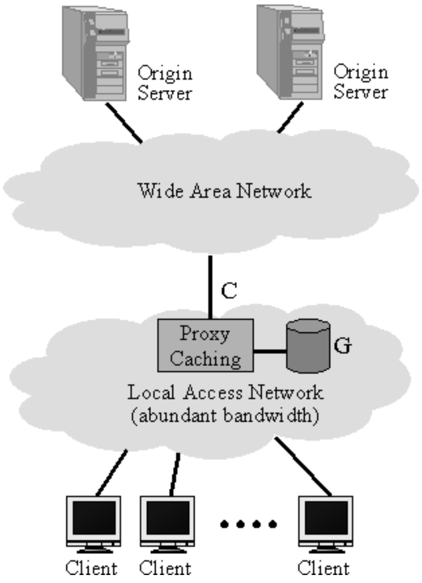


Figure 8.1: Architecture for caching and streaming of layered encoded video.

8.3.1 Layered Video

The continuous media objects available on the origin servers are prerecorded audio and video objects, such as CD–quality music clips, short video clips (e.g., news clips, trailers or music videos) or full–length movies or on–line lectures. Our focus in this study is on video objects that have been encoded using layered (hierarchical) encoding techniques [40,42,86]. With hierarchical encoding each video object is encoded into a base layer and one or more enhancement layers. The base layer contains the most essential basic quality information. The enhancement layers provide quality enhancements. A particular enhancement layer can only be decoded if all lower quality layers are available. Therefore, an enhancement layer is useless for the client if the corresponding lower quality layers are not available.

Layered video allows service providers to offer flexible streaming services to clients with vastly different reception bandwidths and decoding capabilities. Typically, wireless clients and clients with modem–speed wireline Internet access will request only the base layer stream. Clients with high–speed ADSL or cable modem access, on the other hand, may wish to receive higher quality streams consisting of base layer as well enhancement layers. Furthermore, layered video allows for flexible pricing structures. A service provider may offer the base layer stream at a basic rate and charge a premium for the enhancement layers. In other words, clients are charged more when receiving more layers (i.e., higher quality streams). Such a pricing structure might prompt clients to request the cheaper base layer–only stream of a news clip or talk show, say, while requesting the more expensive high quality stream of an entertainment movie.

To make the notion of layered video objects more precise, suppose that there are M video objects. We assume that the video objects are encoded into Constant Bit Rate (CBR) layers, which is a reasonable first approximation of the output of hierarchical codecs. For notational simplicity we assume that all video objects are encoded into L layers. (Our model extends to video objects that differ in the number of layers in a straightforward manner.) Let $r_l(m)$ denote the rate (in bit/sec) of layer l, $l = 1, \ldots, L$, of video object m, $m = 1, \ldots, M$. We define a j–quality stream as a stream consisting of layers $1, 2, \ldots, j$. Let $T(m)$, $m = 1, \ldots, M$, denote the length (in seconds) of video object m. Let $R(j, m)$ denote the revenue accrued from providing a j–quality stream of object m.

8.3.2 Proxy Server

The proxy server is located close to the clients. It is connected to the origin servers via a wide area network (e.g., the Internet). We model the bandwidth available for streaming continuous media from the origin servers to the proxy server as a bottleneck link of fixed capacity C (bit/sec). The proxy is connected to the clients via a local access network. The local access network could be a LAN running over Ethernet, or a residential access network using xDSL or HFC technologies. For the purpose of this study we assume that there is abundant bandwidth for continuous media streaming from the proxy to the clients. We model the proxy server as having a storage capacity of G (bytes). We assume that the proxy storage has infinite storage bandwidth (for reading from storage). We note that the proxy storage is typically a disk array with limited storage bandwidth due to the limited disk bandwidths and seek and rotational overheads. Our focus in this study, however, is on gaining a fundamental understanding of the impact of the two basic streaming resources (bottleneck bandwidth C and cache space G) on the proxy performance. We refer the interested reader to [9, 27, 64] for a detailed discussion of the disk array limitations as well as discussions on replication and striping techniques to mitigate these limitations.

We consider a caching scenario where the cache contents are updated periodically, say every few hours, daily, or weekly. The periodic cache updates are based on estimates of the request pattern of the proxy's client community. A service provider may estimate the request pattern from observations over the last couple of days or weeks. Suppose that the requests for video streams arrive according to a Poisson process with rate λ (requests/sec). Let $p(j, m)$ denote the popularity of the j–quality stream of object m, that is, $p(j, m)$ is the probability that a request is for the j–quality stream of object m. These popularities could be estimated from the observed requests using an exponential weighted moving average. As a proper probability mass distribution the $p(j, m)$'s satisfy $\sum_{m=1}^{M} \sum_{j=1}^{L} p(j, m) = 1$. Also, note that the arrival rate of requests for the j–quality stream of object m is given by $\lambda p(j, m)$.

Our focus in this study is on caching strategies that cache complete layers of video objects in the proxy. Our goal is to cache object layers so as to maximize the revenue accrued from the streaming service. When updating the cache our heuristics give layers of very popular objects priority over layers of moderately popular objects. Moreover, lower quality layers are given priority over higher quality layers (as these require the lower quality layers for decoding at the clients).

To keep track of the cached object layers we introduce a vector of cache indicators $\mathbf{c} = (c_1, c_2, \ldots, c_M)$, with $0 \leq c_m \leq L$ for $m = 1, \ldots, M$. The indicator c_m is set to i if layers 1 through i of object m are cached. Note that $c_m = 0$ indicates that no layer of object m is cached. With the cache indicator notation the cache space occupied by the cached object layers is given by

$$S(\mathbf{c}) = \sum_{m=1}^{M} \sum_{l=1}^{c_m} r_l(m)T(m). \tag{8.1}$$

8.3.3 Stream Delivery

The client directs its request for a j–quality stream of a video object m to its proxy server (for instance by using the Real Time Streaming Protocol (RTSP) [77]). If all the requested layers are cached in the proxy ($c_m \geq j$), the requested layers are streamed from the proxy over the local access network to the client. If layers are missing in the proxy ($c_m < j$), the appropriate origin server attempts to establish a connection for the streaming of the missing layers $c_m + 1, \ldots, j$ at rate $\sum_{l=c_m+1}^{j} r_l(m)$ over the bottleneck link to the client. If there is sufficient bandwidth available, the connection is established and the stream occupies the link bandwidth $\sum_{l=c_m+1}^{j} r_l(m)$ over the lifetime of the stream. (The layers $1, \ldots, c_m$ are streamed from the proxy directly to the client.) We assume that the client watches the entire stream without interruptions, thus the bandwidth $\sum_{l=c_m+1}^{j} r_l(m)$ is occupied for $T(m)$ seconds. In the case there is not sufficient bandwidth available on the bottleneck link, we consider the request as blocked. (In Section 8.5 we study a refined model where clients may settle for a lower quality stream in case their original request is blocked.)

Formally, let $B_{\mathbf{c}}(j, m)$ denote the blocking probability of the request for a j–quality stream of object m, given the cache configuration $\mathbf{c}$. Clearly, there is no blocking when all requested layers are cached, that is, $B_{\mathbf{c}}(j, m) = 0$ for $c_m \geq j$. If the request requires the streaming of layers over the bottleneck link ($c_m < j$), blocking occurs with a non–zero probability $B_{\mathbf{c}}(j, m)$. We calculate the blocking probabilities $B_{\mathbf{c}}(j, m)$ using results from the analysis of multiservice loss models [72].

In this appendix we give an overview of the calculation of the blocking probabilities $B_{\mathbf{c}}(j, m)$, which are non–zero for $c_m < j$. We calculate the blocking probabilities using results from the analysis of multiservice loss models. We refer the interested reader to [72] for a detailed discussion of this analysis. We model the bottleneck link for continuous media streaming from the origin servers to the proxy server as a stochastic knapsack of capacity C. We model requests for j–quality streams of object m as a distinct class of requests. Let $\boldsymbol{b}_{\mathbf{c}} = (b_{\mathbf{c}}(j, m))$, $m = 1, \ldots, M$, $j = 1, \ldots, L$, be the vector of the sizes of the requests. Note that this vector has ML elements. Recall that a request for a j–quality stream of object m of which the c_m–quality stream is cached requires the bandwidth $\sum_{l=c_m+1}^{j} r_l(m)$ on the bottleneck link; hence $b_{\mathbf{c}}(j, m) = \sum_{l=c_m+1}^{j} r_l(m)$ for $c_m < j$ and $b_{\mathbf{c}}(j, m) = 0$ for $c_m \geq j$. Without loss of generality we assume that C and all $b_{\mathbf{c}}(j, m)$'s are positive integers. Let $\boldsymbol{n} = (n(j, m))$, $m = 1, \ldots, M$, $j = 1, \ldots, L$, be the vector of the numbers of $b_{\mathbf{c}}(j, m)$–sized objects in the knapsack. The $n(j, m)$'s are non–negative integers. Let $\mathcal{S}_{\mathbf{c}} = \{\boldsymbol{n} : \boldsymbol{b}_{\mathbf{c}} \cdot \boldsymbol{n} \leq C\}$ be the state space of the stochastic knapsack, where $\boldsymbol{b}_{\mathbf{c}} \cdot \boldsymbol{n} = \sum_{m=1}^{M} \sum_{j=1}^{L} b_{\mathbf{c}}(j, m) n(j, m)$. Furthermore, let $\mathcal{S}_{\mathbf{c}}(j, m)$ be the subset of states in which the knapsack (i.e., the bottleneck link) admits an object of size $b_{\mathbf{c}}(j,m)$ (i.e., a stream of rate $\sum_{l=c_m+1}^{j} r_l(m)$). We have $\mathcal{S}_{\mathbf{c}}(j,m) = \{\boldsymbol{n} \in \mathcal{S}_{\mathbf{c}} : \boldsymbol{b}_{\mathbf{c}} \cdot \boldsymbol{n} \leq C - b_{\mathbf{c}}(j,m)\}$ The blocking probabilities can be explicitly expressed as

$$B_{\mathbf{c}}(j,m) = 1 - \frac{\sum_{\boldsymbol{n} \in \mathcal{S}_{\mathbf{c}}(j,m)} \prod_{m=1}^{M} \prod_{j=1}^{L} (\rho(j,m))^{n(j,m)} / (n(j,m))!}{\sum_{\boldsymbol{n} \in \mathcal{S}_{\mathbf{c}}} \prod_{m=1}^{M} \prod_{j=1}^{L} (\rho(j,m))^{n(j,m)} / (n(j,m))!}, \tag{8.2}$$

where $\rho(j,m) = \lambda p(j,m)T(m)$. Note that $\rho(j,m)$ is the load offered by requests for j–quality streams of object m. The blocking probabilities can be efficiently calculated using the recursive Kaufman–Roberts algorithm [72, p. 23]. The time complexity of the algorithm is $O(CML)$. The complexity is linear in the bandwidth C of the bottleneck link and the number of objects M, which can be huge. The complexity is also linear in the number of encoding layers L, which is typically small (2 – 5).

The expected blocking probability of a client's request is given by

$$B(\mathbf{c}) = \sum_{m=1}^{M} \sum_{j=1}^{L} p(j,m) B_{\mathbf{c}}(j,m).$$

The service provider should strive to keep the expected blocking probability acceptably small, say, less than 5%. The throughput of requests for j–quality streams of object m, that is, the long run rate at which these requests are granted and serviced is $\lambda p(j,m)(1 - B_{\mathbf{c}}(j,m))$. The long run rate of revenue accrued from the serviced j–quality streams of object m is the revenue per served request, $R(j,m)$, multiplied by the throughput. Thus, the long run total rate of revenue of the streaming service is

$$R(\mathbf{c}) = \lambda \sum_{m=1}^{M} \sum_{j=1}^{L} R(j,m) p(j,m)(1 - B_{\mathbf{c}}(j,m)). \tag{8.3}$$

Our goal is to cache object layers so as to maximize the total revenue rate.

8.4 Optimal Caching

In this section we study optimal caching strategies. Suppose that the stream popularities $(p(j,m))$ and the stream characteristics (layer rates $r_l(m)$ and lengths $T(m)$) are given. The question we address is how to best utilize the streaming resources — bottleneck bandwidth C and cache space G — in order to maximize the revenue. Our focus in this study is on optimal caching strategies, that is, we focus on the question: which objects and which layers thereof should be cached in order to maximize the revenue? Formally, we study the optimization problem $\max_{\mathbf{c}} R(\mathbf{c})$ subject to $S(\mathbf{c}) \leq G$. Throughout this study we assume the complete sharing admission policy for the bottleneck link, that is, a connection is always admitted when there is sufficient bandwidth. We note that complete sharing is not necessarily the optimal admission policy. In fact, the optimal admission policy may block a request (even when there is sufficient bandwidth) to save bandwidth for more profitable requests arriving later. We refer the interested reader to [72, Ch. 4] for a detailed discussion on optimal admission policies. Our focus in this study is on the impact of the *caching* policy on the revenue; we assume complete sharing as a baseline admission policy that is simple to describe and administer.

The maximization of the long run revenue rate $R(\mathbf{c})$ over all possible caching strategies (i.e., cache configurations $\mathbf{c}$) is a difficult stochastic optimization problem, that — to the best of our knowledge— is analytically intractable. To illustrate the problem consider a scenario where all video layers have the same rate r and length T, i.e., $r_l(m) = r$ and $T(m) = T$ for all $l = 1, \ldots, L$, and all $m = 1, \ldots, M$. In this scenario all object layers have the size rT. Thus, we can cache up to $G/(rT)$ object layers (which we assume to be an integer for simplicity). Suppose that during the observation period used to estimate the stream popularities, the proxy has recorded requests for M distinct objects from its client community. Thus, there are a total of ML object layers to choose from when filling the cache (with "hot" new releases there might even be more objects to consider). Typically, the cache can accommodate only a small subset of the available object layers, i.e., $G/(rT) \ll ML$. For an exhaustive search there are $\begin{pmatrix} ML \\ G/(rT) \end{pmatrix}$ possibilities to fill the cache completely; a prohibitively large search space even for small ML.

Recall that with layered encoded video a particular enhancement layer can only be decoded if all lower quality layers are available. Therefore, a reasonable restriction of the search space is to consider a particular enhancement layer for caching only if all lower quality layers of the corresponding object are cached. Even the "reasonable" search space, however, is prohibitively large for moderate ML; with $M = 50$, $L = 2$, $G/(rT) = 20$, for instance, there are $2.929 \cdot 10^{16}$ possibilities to fill the cache completely.

Because the maximization problem $\max_{\mathbf{c}} R(\mathbf{c})$ subject to $S(\mathbf{c}) \leq G$ is analytically intractable and exhaustive searches over $\mathbf{c}$ are prohibitive for realistic problems, we propose heuristics for finding the optimal cache composition $\mathbf{c}$.

8.4.1 Utility Heuristics

The basic idea of our utility heuristics is to assign each of the ML object layers a cache utility $u_{l,m}$, $l = 1, \ldots, L$, $m = 1, \ldots, M$. The object layers are then cached in decreasing order of utility, that is, first we cache the object layer with the highest utility, then the object layer with the next highest utility, and so on. If at some point (as the cache fills up) the object layer with the next highest utility does not fit into the remaining cache space, we skip this object layer and try to cache the object layer with the next highest utility. Once a layer of an object has been skipped, all other layers of this object are ignored as we continue "packing" the cache. We propose a number of definitions of the utility $u_{l,m}$ of an object layer; see Table 8.1 for an overview.

The popularity utility is based exclusively on the stream popularities; it is defined by $u_{l,m} = p(l, m) + p(l + 1, m) + \cdots + p(L, m)$. This definition is based on the decoding constraint of layered encoded video, that is, an object layer l is required (i.e., has utility) for

Popularity utility	$u_{l,m} = \sum_{j=l}^{L} p(j,m)$
Revenue utility	$u_{l,m} = \sum_{j=l}^{L} R(j,m)p(j,m)$
Revenue density utility	$u_{l,m} = \sum_{j=l}^{L} \frac{R(j,m)p(j,m)}{r_j(m)T(m)}$

Table 8.1: Utility definitions.

providing l–quality streams (consisting of layers 1 through l), $l+1$–quality streams, ..., and L–quality streams. Note that $u_{l,m}$ is the probability that a request involves the streaming of layer l of object m. Also, note that by definition $u_{l,m} \geq u_{l+1,m}$ for $l = 1, \ldots, L-1$. This, in conjunction with our packing strategy ensures that a particular enhancement layer is cached only if all corresponding lower quality layers are cached.

8.4.2 Evaluation of Heuristics

In this section we present some numerical results from experiments to evaluate various aspects of the heuristics algorithms. We ran two different types of experiments. The bulk of the experiments was carried out analytically, by calculating the revenue according to equation (8.3) and calculating the blocking probabilities as described in the Appendix. All of the results presented in this section are obtained in this fashion. We refer to these experiments as analytical experiments.

We also implemented a cache simulator, in order to study queuing of requests and partial caching. These results are presented in Sections 8.5, 8.6, and 8.7. We refer to these experiments as simulation experiments.

We assume that there are 1000 different movies, each encoded into two layers. The characteristics of each movie are defined by the rate for each layer and its length. The rate for each layer is drawn randomly from a uniform distribution between 0.1 and 3 Mbps, while the length of the movie is drawn from an exponential distribution with an average length of 1 hour.

In all of our experiments client requests arrive according to a Poisson process. The average request arrival rate is 142 Erlangs. The client can request either a base layer only or a complete movie. The request type and the movie requested are drawn randomly from a Zipf distribution with a parameter of $\zeta = 1.0$. The revenue for each movie layer is uniformly distributed between 1 to 10.

The results of interest will be the revenue per hour and the blocking probabilities. To obtain the results with 99% confidence intervals, we run the experiments with different random seeds and we require a minimum of 10000 runs before calculating the confidence intervals. In each run we randomly assign the popularities of movies from the Zipf distri-

	Small Link	
Utility heuristic	**Small Cache**	**Large Cache**
Popularity	1.6%	2.4%
Revenue	2.8%	0.4%
Revenue density	0.3%	0.3%

	Large Link	
Utility heuristic	**Small Cache**	**Large Cache**
Popularity	0.006%	0%
Revenue	0.1%	0%
Revenue density	0.1%	0%

Table 8.2: Average error of heuristics in small problems

bution, the rates and the lengths of the movie layers. The results are calculated as the average value of the revenue per hour from all the runs until the confidence intervals are reached.

We first tested the performance of our heuristics in small problems in order to be able to compare the heuristic against the "reasonable" exhaustive search. For the small problems we set $M = 10$ with each movie having two layers. We varied the link bandwidth C between 3 and 15 Mbit/s and the cache capacity between 3 and 7 Gbytes. The cache could therefore store on the average between 3.5 and 7.6 layers out of the total 20 layers, or between 23.1 and 41.7% of the total movie data.

The results of the small problems are shown in Table 8.2. In Table 8.2 we show the average error obtained with each heuristic compared to the "reasonable" exhaustive search for four different cache configurations. The *Small Link* and *Large Link* refer to link capacities of 3 Mbit/s and 15 Mbit/s, respectively, and *Small Cache* and *Large Cache* refer to 3 Gbyte and 7 Gbyte caches, respectively.

As we can see, our heuristics achieve performance very close to the optimum in most cases. Only when both the link and the cache are small is there any marked difference in performance. This is largely due to the small link capacity, only 3 Mbit/s, which allows us to stream only one movie on the average. As both the link and cache grow in size, we can achieve the same performance as the optimal caching strategy.

To test the performance of our heuristics in real-world size problems, we ran the heuristics for 1000 movies. We varied the cache size between 12 and 560 Gbytes. The cache could therefore hold on the average between 13.9 and 625 layers, or between 0.9 and 41.7% of the total movie data. Given the average length of a movie T_{avg}, the average rate of a movie r_{avg}, and the client request rate λ, we would need on the average $T_{avg} r_{avg} \lambda$ Mbit/s of bandwidth to stream all the requested movies. We varied the link capacity between 10 and 150 Mbit/s, or between 1 and 15% of the total bandwidth required.

Because running the exhaustive search was not feasible for problems this large, we approximated the best possible performance by calculating the revenue when the blocking

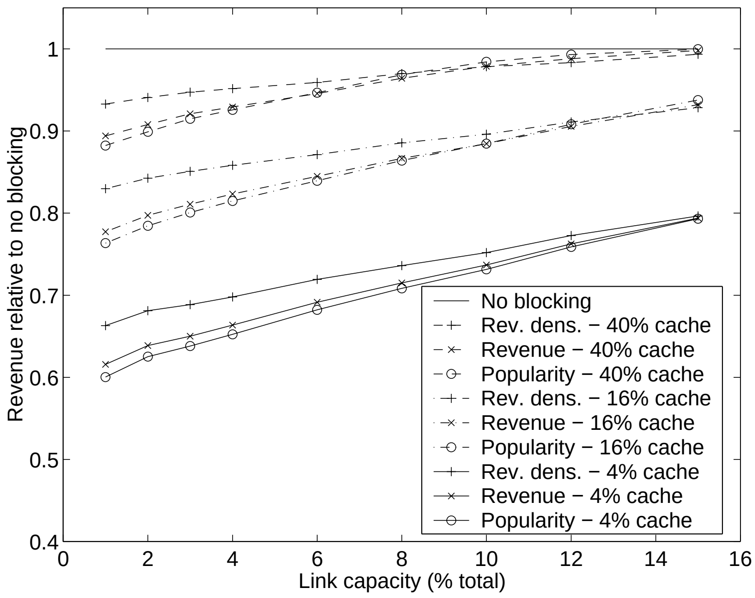


Figure 8.2: Revenue as function of link capacity for 3 different cache sizes

probability was zero. This means that all client requests are always satisfied and it provides us with an upper limit on the achievable revenue. In reality, this upper limit is not reachable unless the link and cache capacities are sufficiently large to ensure that no client requests are ever blocked. In our tests the smallest observed blocking probabilities were around 0.005%.

In Figure 8.2 we show the revenue relative to the no blocking case obtained with 3 different cache sizes as a function of the link capacity. We can see that the revenue density heuristic performs the best overall and that the performance difference is biggest when the link capacity is smaller. As the link capacity increases, the performance difference disappears. We also see that the popularity heuristic has the worst overall performance.

In Figure 8.3 we show the revenue obtained with 2 different link capacities as a function of the cache size. Here the difference between revenue density heuristic and the others is clearer. For example, with a 1% link and a 20% cache (10 Mbit/s link and a cache of 250 Gbytes in our case), revenue density heuristic achieves 87% of the upper limit while the revenue heuristic achieves only 79%. Again, as in Figure 8.2, when we have enough link and cache capacity, the difference between the heuristics disappears. To illustrate the tight confidence intervals we observed, we plot the revenue density heuristic in the 1% link case with the 99% confidence intervals.

Overall, we can conclude that the revenue density utility heuristic has the best performance of the three heuristics studied. This is especially true in situations where we have a shortage of one of the resources, link capacity or cache size. This implies that the revenue density heuristic predicts the usefulness of a layer more accurately than the other two heuristics.

In Figure 8.4 we show the revenue obtained with the revenue density heuristic as a

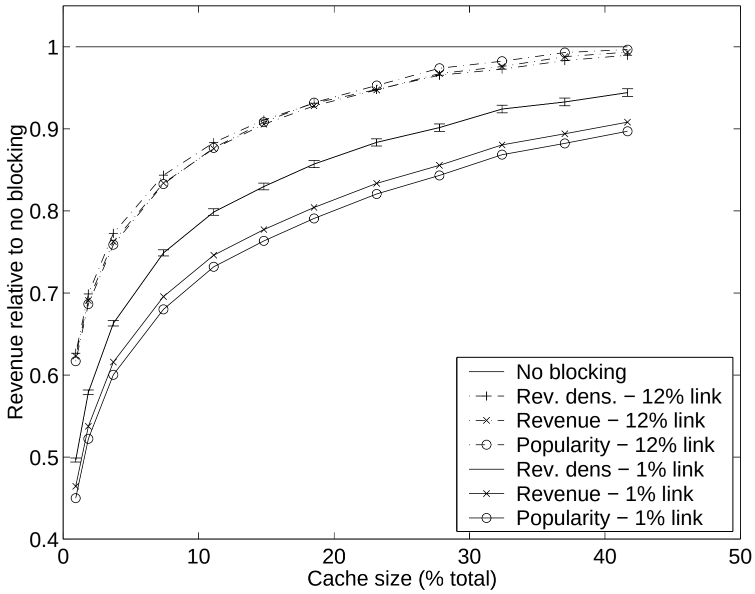


Figure 8.3: Revenue as function of cache size for 2 different link capacities

function of both link capacity and cache size. We observe that if we have a shortage of both resources, we should first increase the cache before increasing the link capacity. We see that when the cache size is around 20% of the total movie data (250 Gbytes in our case), further increase in cache size provides only small gains in revenue. At this point, increasing the link capacity provides larger gains in revenue. This behavior can also be observed in Figures 8.2 and 8.3 where we can see that the revenue increases roughly linearly with the link capacity and roughly logarithmically with the cache size.

In Figure 8.5 we show the expected blocking probability for the revenue density heuristic. Note that the plot shows $1 - B(\mathbf{c})$ and smallest expected blocking probability is therefore obtained when the curve is close to 1. This plot reflects the typical blocking probabilities we obtained in all of our experiments, including the experiments in Sections 8.5, 8.6, and 8.7.

We also studied the effects of varying the parameter ζ in the Zipf-distribution and varying the client request rate, λ. Previous studies in Web caching and server access dynamics have found that ζ can vary from 0.6 in Web proxies [10] up to 1.4 in popular Web servers [59]. We studied four different values of ζ, namely 0.6, 0.8, 1.0, and 1.3. In Figure 8.6 we show the revenue obtained with each of the four parameter values for three different link capacities as a function of the cache size. We can see that the curves corresponding to one value of ζ are close together and that there is a significant difference in groups of curves belonging to different values of ζ. This implies that a decrease in ζ (movies become more equally popular) requires significant increases in link capacity and cache size to keep the revenue at the same level. On the other hand, should ζ increase (small number of movies become very popular), we can achieve the same revenue with considerably less resources.

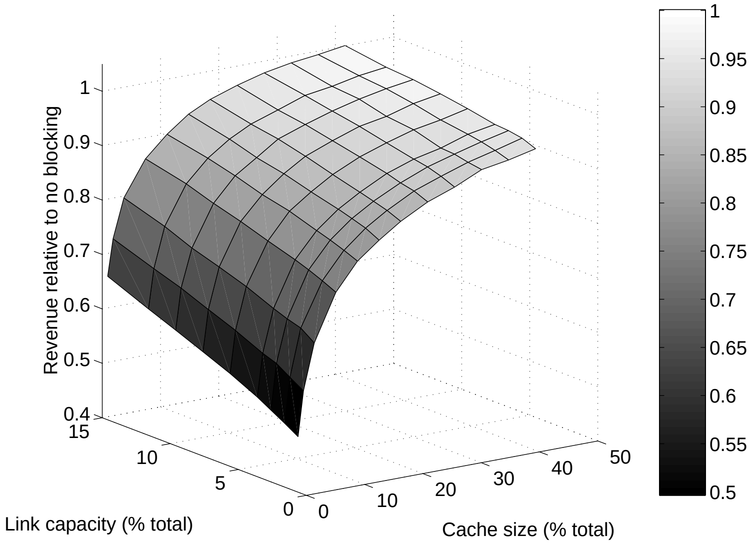


Figure 8.4: Revenue as function of cache size and link capacity

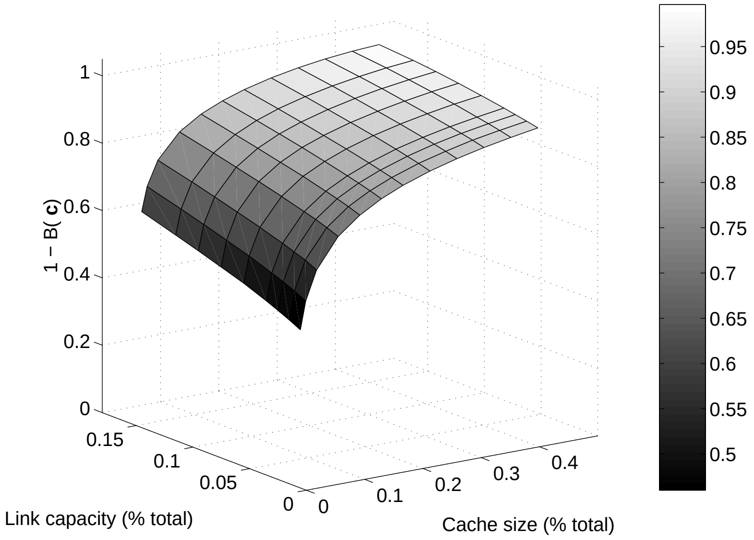


Figure 8.5: $1 - B(\mathbf{c})$ as function of cache size and link capacity

In Figure 8.7 we show the effects of varying the client request rate. We plot curves for three different values of λ for two different link capacities. The curves for "Low λ at 6% link" and "Medium λ at 10% link" fall on top of each other. We can clearly see that the client request rate has much less effect on the revenue than the Zipf-parameter. In some cases, it is possible to counter the changes in request rate by increasing the link capacity or cache size. For example, if the request rate goes from Low to Medium, increasing the link capacity from 6% to 10% (60 Mbit/s to 100 Mbit/s in this case) keeps the revenue the same.

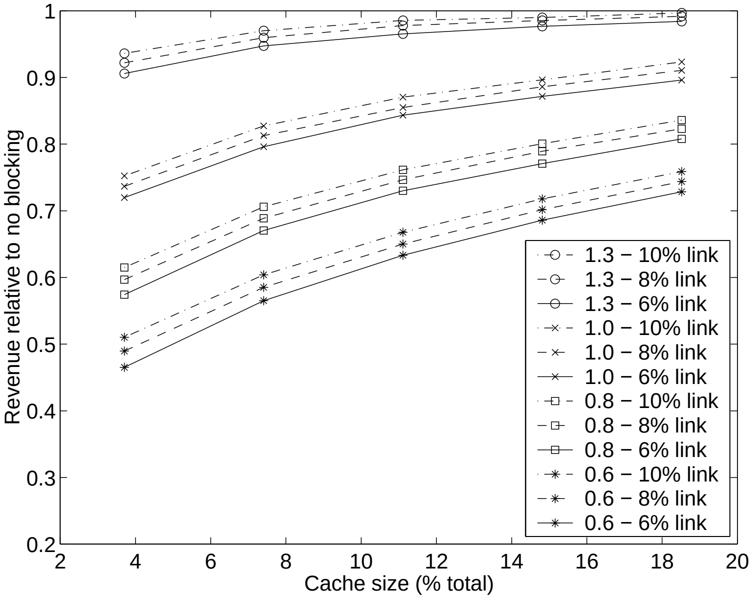


Figure 8.6: Effect of Zipf-parameter ζ on revenue

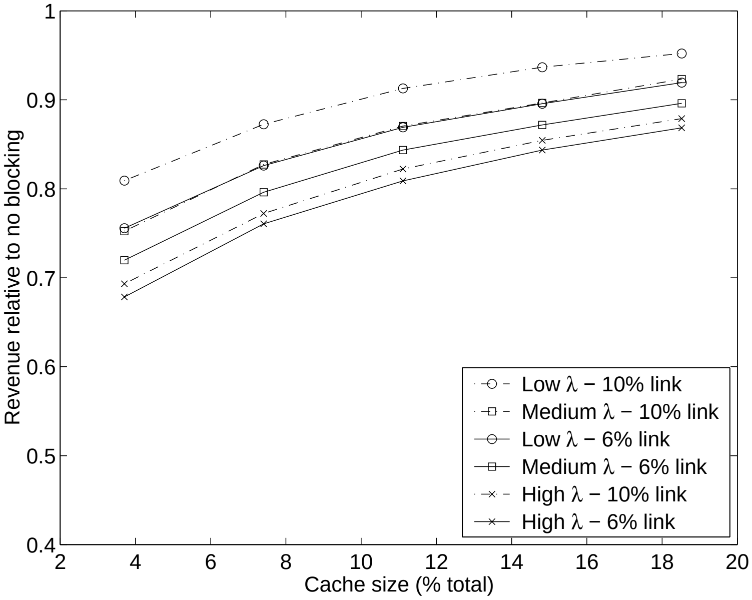


Figure 8.7: Effect of client request rate on revenue

In conclusion, all three of our heuristics perform well under many different link and cache size combinations. The revenue density heuristic achieves the best performance under constrained conditions.

8.5 Negotiation about Stream Quality

In this section we study a negotiation scheme where in case the client's original request is blocked, the service provider tries to offer a lower quality stream of the requested object. The client may then settle for this lower quality stream. The question we address is: how much additional revenue is incurred with this "negotiation." As we shall demonstrate, this intuitively quite appealing approach adds very little to the revenue in most situations. For simplicity we focus in this section on video objects that are encoded into L =2 layers: a base layer and one enhancement layer. (Our arguments extend to the case of more encoding layers in a straightforward manner.) Suppose that a client requests a 2–quality stream (consisting of base layer and enhancement layer) of object m. Suppose that the cache configuration is given by $\mathbf{c}$. Clearly, the original request can only be blocked if not all requested layers are cached, that is, if $c_m < 2$. If the client's original request for a 2–quality stream of object m is blocked the service provider tries to offer a 1–quality (i.e., base layer) stream of the object. The service provider is able to make this offer if the base layer stream is not blocked.

Note that the negotiations increase the arrival rates of requests for base layer streams. This is because the blocked 2–quality stream requests "reappear" as base layer stream requests. With negotiations the arrival rates of base layer stream requests depend on the blocking probabilities of 2–quality stream requests, that is, the system becomes a generalized stochastic knapsack [72, Ch. 3]. Calculating the blocking probabilities of the generalized stochastic knapsack, however, is quite unwieldy. Therefore we approximate the blocking probabilities of the streaming system with negotiations. In typical streaming systems the blocking probabilities are small, typically less than 5 %. The increase in the arrival rates of base layer stream requests is therefore relatively small. We approximate the blocking probabilities of the system with negotiations by the blocking probabilities of the system without negotiations. The probability that the client's original request for a 2–quality stream of object m is blocked is approximately $B_{\mathbf{c}}(2,m)$. The probability that the corresponding base layer stream is not blocked is approximately $1-B_{\mathbf{c}}(1,m)$. Suppose that the client accepts the quality degradation with probability $P_{\text{acc}}(m)$. If the client does not accept the offer the negotiation terminates. Thus, given that the negotiation is entered, it ends in a success (i.e., service provider and client settle for a base layer stream) with probability $(1-B_{\mathbf{c}}(1,m))P_{\text{acc}}(m)$. The long run rate (successful negotiations per hour) at which negotiations settle for a base layer stream of object m is $\lambda p(2,m)B_{\mathbf{c}}(2,m)(1-B_{\mathbf{c}}(1,m))P_{\text{acc}}(m)$. Suppose that each successful negotiation resulting in the delivery of a base layer stream of object m incurs a revenue of $R_{\text{neg}}(1,m)$ (which may be different from $R(1,m)$ as the service provider may offer the base layer at a discount in the negotiation). Thus, the long run total rate of revenue incurred from successful negotiations is

$$R_{\text{neg}}(\mathbf{c}) = \lambda \sum_{m=1}^{M} R_{\text{neg}}(1,m)p(2,m)B_{\mathbf{c}}(2,m)(1-B_{\mathbf{c}}(1,m))P_{\text{acc}}(m).$$

The long run total rate of revenue of the streaming service with negotiations is $R(\mathbf{c}) + R_{\text{neg}}(\mathbf{c})$, where $R(\mathbf{c})$, the revenue rate incurred from serving first–choice requests, is given by (8.3).

8.5.1 Numerical Results

We experimented with adding the renegotiation revenue to our tests. We first tested the quality of the approximation used in calculating the blocking probability of the system with renegotiation against the results obtained from our cache simulator. We varied the link capacities between 10 to 120 Mbps. Our results show a close approximation of the analytical experiments to the simulation experiments with an average error of 0.4–0.5% for 12 Gbyte cache and 0.7–1.1% for 560 Gbyte cache. The results presented here are from the analytical experiments.

Figure 8.8 shows how much extra revenue renegotiation could bring relative to the baseline revenue $R(\mathbf{c})$. The revenue in Figure 8.8 is based on the assumption that the client will always accept the lower quality version if one is available, i.e., $P_{\text{acc}}(m) = 1$ for $m = 1, \ldots, M$. We also assumed that $R_{\text{neg}}(1, m) = R(1, m)$ for $m = 1, \ldots, M$, i.e., the revenue from the renegotiated stream is the same as if the client had requested the lower quality stream in the first place. These two assumptions give us the maximum possible gain from renegotiation.

As we can see from Figure 8.8, the largest gains from renegotiation are achieved when the cache size is extremely small, only 1–2% of the total amount of data. The renegotiation gains are almost insensitive to link capacity with the exception of very small link capacities where the gains are slightly smaller. The maximum gain we observed is around 20% and the gain drops sharply as the cache size increases. The maximum gain would decrease as the client acceptance probability P_{acc} decreases. Also, if the cache size and link capacity are large, the potential gain from renegotiation is typically well below 1%. We can therefore conclude that renegotiation, although intuitively appealing, does not provide any significant increase in revenue in most situations. This is because renegotiation is only applicable to blocked requests and one of the goals of a cache operator would be to keep the expected blocking probability as low as possible.

8.6 Queueing of Requests

In this section we study a request queueing scheme where in case the client's request is blocked, the service provider queues the request. With the queueing strategies, we expect that the queued requests make use of the resources released by currently served requests. This has the potential of increasing the resource utilization and thus, bringing additional revenue. The question is how much additional revenue does it bring.

To answer this question, we used our cache simulator to obtain the results in this section. To align the experiments with the real-world practice, we assume that a client will cancel its request after waiting for some time, referred to as the *request timeout period*. We model the timeout period using an exponential distribution with an average of 5 minutes.

We assume that the queue is of a finite size and it can hold up to 100 requests. An incoming request finding a full buffer will be blocked. We consider three different strategies

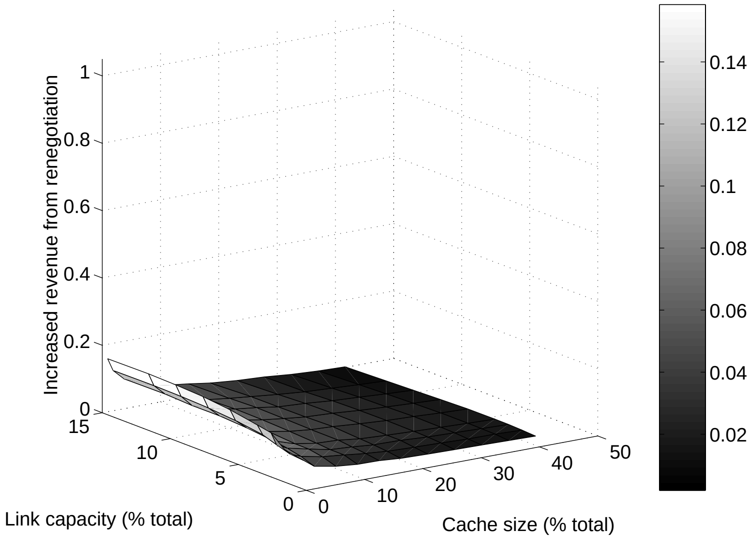


Figure 8.8: Increased revenue from renegotiation

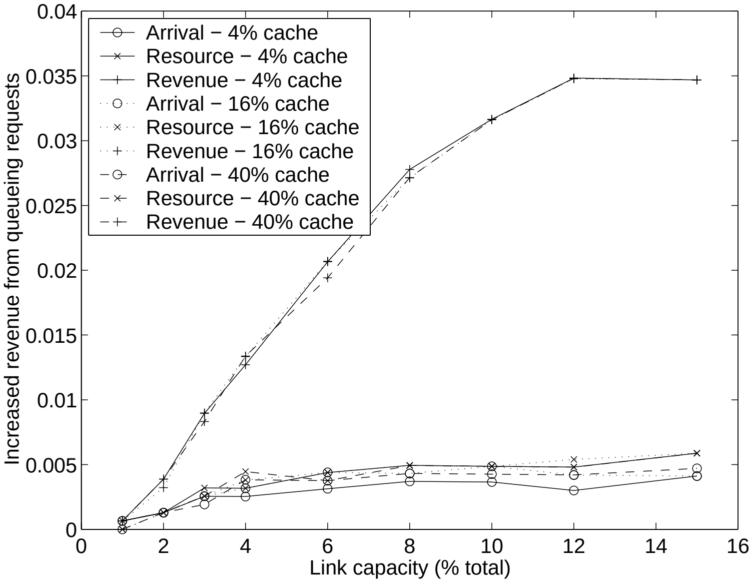


Figure 8.9: Increased revenue from queueing requests for buffer size of 100

for ordering the requests in the queue, i.e., based on the order of request *arrivals*, their required *resources* and the potential *revenues*.

Figure 8.9 shows how much extra revenue queueing of requests could bring relative to the baseline revenue $R(\mathbf{c})$. As we can see from the figure, the gain from introducing the queue is very small. The gain is not affected by the cache size and generally increases with the link capacity.

With the limited bandwidth of the bottleneck link, which causes request blocking in the first place, the serving of one request from the queue will mean the blocking of another incoming request. This results in a near zero gain in the number of requests served. A possible gain can be achieved by changing the request service strategies, for example by serving the request according to the potential revenue that it brings.

8.7 Is Partial Caching Useful ?

Consider a streaming system where *clients are only interested in complete streams* (consisting of all L layers) and *no revenue is incurred for partial streams* (consisting of less than L layers). The question we address is: in such a system is caching of partial streams (e.g., base layers) beneficial? Interestingly, the answer appears to be *no*.

We focus on the homogeneous two–layer case where the video objects are encoded into $L = 2$ layers: a base layer of rate $r_1(m)$ and one enhancement layer of rate $r_2(m)$. For simplicity we assume that (1) all videos have the same layer rates, i.e., $r_1(m) = r_b$ and $r_2(m) = r_e$ for $m = 1, \ldots, M$, and (2) all videos have the same length T. We study a system where clients request only complete streams (consisting of both base layer and enhancement layer), i.e., $p(1, m) = 0$ for $m = 1, \ldots, M$. For ease of notation we write $p(m)$ for $p(2, m)$ and note that $\sum_{m=1}^{M} p(m) = 1$. We order the video objects from most popular to least popular; thus, $p(m) \geq p(m+1),\ m = 1, \ldots, M-1$. In the considered system no revenue is incurred for streams consisting of only the base layer, i.e., $R(1, m) = 0$. We assume that all complete streams incur the same revenue, i.e., $R(2, m) = R$ for $m = 1, \ldots, M$.

We investigate a caching strategy that caches both base and enhancement layer of very popular video objects. For moderately popular objects only the base layer is cached (and the enhancement layer is streamed upon request over the bottleneck link of capacity C). For relatively unpopular objects neither base nor enhancement layer is cached. Let N_1 denote the number of completely cached objects. Clearly, $0 \leq N_1 \leq \lfloor G/(r_b + r_e)T) \rfloor := N_1^{\max}$. Let N_2 denote the number of cached base layers. The N_1 completely cached objects take up the cache space $N_1(r_b + r_e)T$. Hence, $0 \leq N_2 \leq \lfloor (G - N_1(r_b + r_e)T)/(r_b T) \rfloor := N_2^{\max}$. The investigated caching strategy caches base and enhancement layer of the N_1 most popular objects, that is, objects $1, \ldots, N_1$. It caches the base layers of the N_2 next most popular objects, that is of objects $N_1 + 1, \ldots, N_1 + N_2$.

The probability that a request is for a completely cached object is $P_1 = \sum_{m=1}^{N_1} p(m)$. The probability that a request is for an object for which only the base layer has been cached is $P_2 = \sum_{m=N_1+1}^{N_1+N_2} p(m)$. Note that the probability that a request is for an object which has not been cached at all is $P_3 = 1 - P_1 - P_2$.

We model the bottleneck link connecting the cache to the wide area network again as a stochastic knapsack [72]. The bottleneck link is modeled as a knapsack of capacity C. We refer to streams of completely cached video objects as class 1 streams. Class 1 streams consume no bandwidth on the bottleneck link, that is, $b_1 = 0$. The arrival rate of class 1 streams is $\lambda_1 = \lambda P_1$. Streams of video objects for which only the base layer is cached are referred to as class 2 streams. Class 2 streams consume the bandwidth $b_2 = r_e$. The arrival rate for class 2 streams is $\lambda_2 = \lambda P_2$. Streams of video objects which have not been cached at all are referred to as class 3 streams. Class 3 streams consume the bandwidth

$b_3 = r_b + r_e$ and have an arrival rate of $\lambda_3 = \lambda P_3$. All streams have a fixed holding time T.

Our objective is to maximize the total long run revenue rate, or equivalently, the long run throughput of requests (i.e., the long run rate at which requests are granted and serviced). Towards this end let TH_k denote the long run throughput of class k requests. Also, let TH denote the long run total throughput of requests. Clearly, $TH = TH_1 + TH_2 + TH_3$. Let B_k denote the probability that a request for a stream of class k is blocked. Obviously, $B_1 = 0$ since class 1 streams do not consume any bandwidth. Thus, $TH = \lambda[P_1 + P_2(1 - B_2) + P_3(1 - B_3)]$.

8.7.1 Numerical Results

We used our cache simulator to study partial caching. All the results in this and the next section are obtained from the simulator. We used the same experiment setup (layer rates, video lengths, and Zipf-parameter) as for evaluating the performance of the utility heuristics in Section 8.4.2. In fact, we can consider the partial caching case as a special case of the utility heuristics. Note that for the partial caching case the utilities of the base and enhancement layer of a given movie are the same and thus base layer and enhancement layer are cached together.

In our experiments we question the usefulness of partial caching where a portion of the cache is reserved for caching base layers only. Doing so allows us to cache (at least the base layers of) a larger number of movies for the same cache size. An intuitive question to follow is whether *trunk reservation* is beneficial. With trunk reservation a portion of the link bandwidth, say $C_2 = x\%$ of C, $x = 0 - 100$, is reserved for streaming the enhancement layers of the class 2 movies which have base layers in the cache. We naturally expect that a combination of these two strategies may give us the best throughput.

Figure 8.10 shows the normalized throughput as a function of the percentage of cache space used for caching complete movies. The figure also shows the throughput for different link reservation and cache sizes. The link reservation of 0% implies a complete sharing of the link bandwidth between class 2 and class 3 streams. This case can be analyzed using the *stochastic knapsack* formulation, see Section 8.3.3, which gives us the blocking probabilities B_2 and B_3 and hence the throughput. On the other hand, the link reservation of 100% implies a total blocking of class 3 streams. The link is solely used for streaming enhancement layers for class 2 streams which have base layers cached. As we have only one traffic class, this case can be analyzed using the Erlang–B formula with the number of trunks being C/r_e. For the other cases with the link reservations between 0 to 100%, we use simulations to obtain the throughput.

The results confirm our intuition that once the base layers are cached, it is beneficial to reserve some bandwidth to give us an optimum throughput. For example, if we reserve 30% of the cache space for complete movies, which also means that we reserve 70% of the cache for base layers, then reserving any amount of bandwidth for streaming class 2 movies will give us better throughput than complete sharing. However, we can clearly see from Figure 8.10 that, for a given cache size, the maximum is always obtained at the right edge of the plot, that is, when the whole cache is reserved for caching complete movies. In this

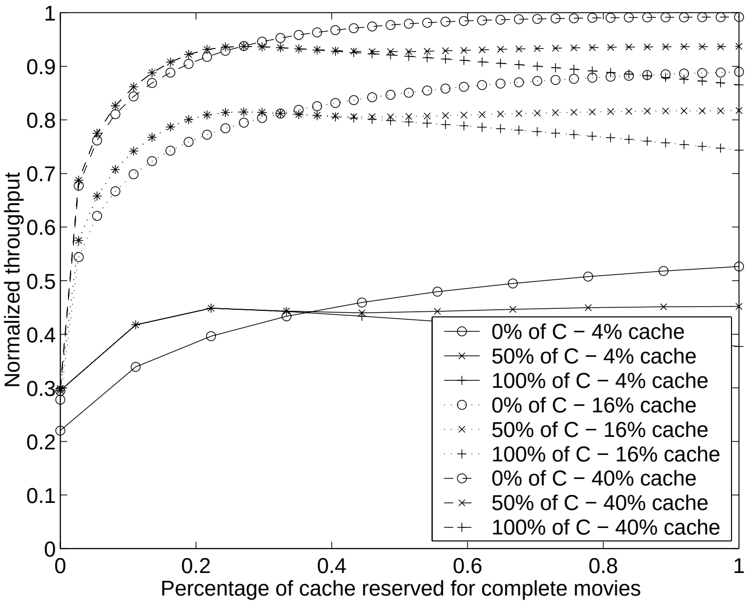


Figure 8.10: Normalized throughput for partial caching and trunk reservation with $C = 150$ Mbps

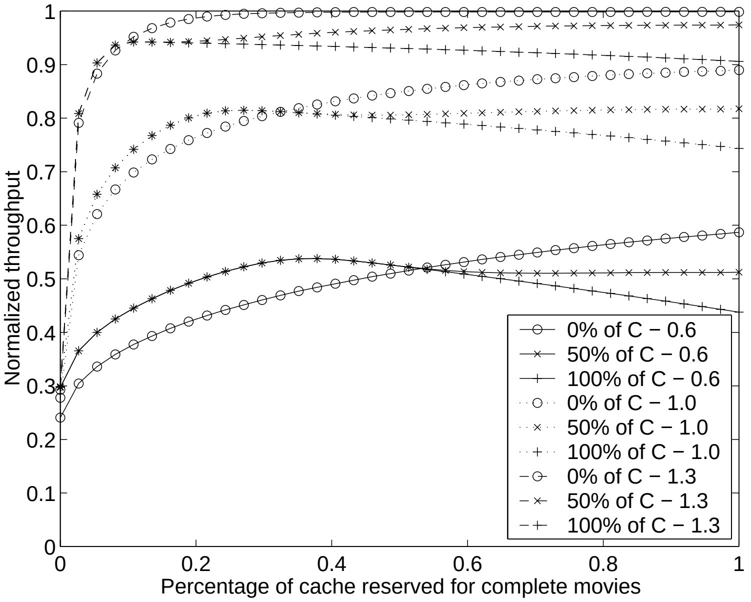


Figure 8.11: Normalized throughput for partial caching and trunk reservation with different Zipf parameters

case, there are no class 2 streams and thus, the link is used exclusively for streaming the class 3 movies.

Figure 8.11 shows the effect of varying the popularity of the movies. We observe that the proportion of the cache space that needs to be reserved to achieve the optimum throughput for the partial caching case changes with the Zipf parameter. This makes it harder to dimension the cache properly to achieve the optimum throughput at all times. Considering this difficulty and the fact that reserving the entire cache for caching complete movies give the maximum throughput, our experiments indicate that the partial caching is not beneficial.

8.8 Related Work

There are only few studies on distributing video objects with caches, all of which are complementary to the issues studied in this chapter. Rejaie *et al.* propose a proxy caching mechanism [67] in conjunction with a congestion control mechanism [65, 66] for layered–encoded video. The basic idea of their caching mechanism is to cache segments of layers according to the objects' popularities: the more popular an object, the more complete are the individual layers cached and the more layers are cached (partially). When streaming an object to a client, the layer segments that are not cached at the proxy are obtained from the origin server.

A related idea is explored by Wang *et al.* in their study on video staging [88]. With video staging the part of the VBR video stream, that exceeds a certain cut–off rate (i.e., the bursts of a VBR stream) is cached at the proxy while the lower (now smoother) part of the video stream is stored at the origin server.

Sen *et al.* [78] propose to cache a prefix (i.e., the initial frames) of video streams at the proxy and to employ work–ahead smoothing while streaming the object from the proxy to the client. The cached prefix hides the potentially large initial start–up delay of the work–ahead transmission schedule from the client.

Tewari *et al.* [85] propose a Resource Based Caching (RBC) scheme for video objects encoded into one CBR layer. They model the cache as a two resource (storage space and bandwidth) constrained knapsack and study replacement policies that take the objects' sizes as well as CBR bandwidth into account. The replacement policies are evaluated through simulations. Our work differs from RBC in that we develop an *analytical* stochastic knapsack model for the two resource problem. Moreover, we analyze a streaming system where videos are encoded into multiple layers.

8.9 Conclusion

In this chapter we have formulated an analytical stochastic knapsack model for the layered video caching problem. We have proposed three different heuristics for determining which layers of which videos to cache. Through extensive numerical experiments we have found that all our heuristics perform well and that the best performance is obtained with the revenue density heuristic. Our heuristics are useful for cache operators in both provisioning the caching system as well as deciding on-line the gain from caching a given layer of a given video. To the best of our knowledge, this is the first study to consider an analytical model of this 2-resource problem.

We also considered two intuitive extensions, renegotiation and queueing of requests, but found that they provide little extra gain to the cache operator. As a special case we considered a situation where clients only request complete video streams. Our results indicate that in this special case, best performance is obtained if videos are cached completely.

There are also a number of avenues for future research, such as considering dynamically changing request patterns. Furthermore, there are a number of special scenarios where theoretical results may be obtainable.

Chapter 9

Conclusion

This thesis has investigated content distribution on the Internet. In this chapter we will present a summary of the contributions of this thesis and outline possible directions for future work.

9.1 Summary

In the first part of this thesis, Chapters 3–5, we have studied client redirection in content distribution architectures. We have presented the Location Data System (LDS) which allows clients to locate copies of objects that are stored on different object servers on the Web. LDS is a small extension to the existing Domain Name System (DNS) and when a client wants to locate an object, it performs a query on the whole URL and receives as a response all the object servers which have a copy of the requested object. The LDS system allows for incremental deployment and it can be fine-tuned to reduce the number of query messages sent into the network.

We have presented an architecture for an Internet-wide replicated directory service. As an example of our architecture, we have shown how the current DNS system could be implemented with the replicated architecture. The key feature of the new architecture is that it allows us to store rapidly changing information and guarantee that the data is coherent in the whole network. In addition, our architecture can significantly speed up the DNS queries. Our architecture can be deployed incrementally and one of its key features is that it requires no changes to existing DNS software, nor does it change the syntax or semantics of any existing DNS messages. We also perform an extensive performance evaluation to determine suitable values for deciding how long can we allow nameservers to cache the information obtained through our architecture.

As our third contribution in the first part of the thesis, we have performed an extensive numerical evaluation of the performance of the client redirection mechanisms used by modern content distrbution networks. Modern CDNs use DNS to redirect clients using either full or selective redirection. Using extensive simulation experiments we have quantified the cost of having to set up new TCP connections, as required by the selective redirection mechanism, and our results have shown that this cost is typically extremely high. We have also verified these simulation results with experiments on live servers on the Internet and we have observed similar results.

In the second part of this thesis, Chapters 6–8, we have studied object replication in content distribution architectures. We have studied object replication strategies in content distribution networks. We formulated the problem of optimally placing object replicas as a combinatorial optimization problem which we have shown to be NP-complete. We have developed several different heuristics to try to place objects in a CDN and we have evaluated their performance on real Internet AS-level topologies. We have found that heuristics which take into account the whole state of the network across all ASes yield superior performance to heuristics which try to act based only on information available locally to a single AS. We have also considered cooperation in peer-to-peer networks. We have developed a model for evaluating the benefits of cooperation between the peer-to-peer users in different ASes. Our model also provides a simple mechanism to test if cooperation would be beneficial. Our results have shown that cooperation between neighboring ASes typically yields high benefits which diminish as the distance between the cooperating ASes increases.

As the second contribution in the second part of the thesis, we have considered optimal content replication in peer-to-peer communities. We have distinguished between two types of communities: content caches and content clubs. We have modeled the problem of optimally replicating content in a community as an integer programming problem. The solution of this problem provides us with an upper bound on hit-rate that can be achieved by any algorithm. We have developed adaptive algorithms that replicate objects on-the-fly and have compared the hit-rates with the upper bound. We have found that our algorithms combined with a distributed least-frequently-used replacement policy provide near-optimal hit-rates and replica profiles. We have also provided a formulation of the initial integer programming problem which takes into account hot-spots and balances the load evenly across all peers.

As the final contribution of this thesis, we have investigated the distribution of layered encoded video through caches. We considered the two-resource problem of cache storage and network link capacity which we modeled using a stochastic knapsack model. We proposed three heuristics to determine which layers of which videos we should cache to maximize the accrued revenue. Through extensive numerical experiments we have found that all our heuristics perform well and that the best performance is obtained with the revenue density heuristic. A cache operator could use the heuristics we have developed to determine on-line the utility of caching a given layer of a given video. We have also considered two intuitive extensions to the initial model: renegotiation of quality and queueing of requests. Our results indicate that these extensions do not provide any significant gains in overall revenue. Finally, we have considered the problem where clients are interested only in complete videos. We have found that in this case, caching only some layers is not beneficial and we should cache all layers of a video together.

9.2 Directions for Future Work

This work has several possible avenues for future work. Each of the chapters outlines ways in which the work presented in that chapter can be extended. The field of content distribution and especially peer-to-peer networking holds much promise for future research. Below we outline how the research in this thesis could be extended in this direction.

The new, emerging wireless and mobile ad-hoc networks are well suited for a peer-to-peer-like content distribution system. The transient nature of these networks will require new protocols and architectures for ensuring efficient content delivery, both for static content and streaming media. These kinds of networks are likely to become commonplace in the next years and they need to be studied. Accompanying this research, we also need to build prototypes to test the new protocols and architectures in practice.

Some key issues in such networks are security, privacy, and object naming. Current peer-to-peer networks do not offer any protection against malicious users trying to inject false content or to deny service from other users. In an environment where users rely heavily on others, such as a wireless ad-hoc network, these issues must be resolved.

Also, in a network where all nodes participate in its operation, we need to ensure that the privacy requirements are met. These requirements can depend on the application or the user's preferences. Some users may even wish to remain completely anonymous; the network should offer this possibility.

We also need efficient methods for finding objects in these networks. This requires both an efficient naming scheme as well as query methods for finding content. In a network where content freely migrates from one node to another, naming schemes which tie the object into a location, such as URLs, are inappropriate. New naming schemes need to be developed and their performance evaluated. Query mechanisms for finding the content are closely tied to the naming system and their performance is critical to the overall performance of the system. Hence, they need to be studied closely in order to avoid them becoming the bottleneck of the system.

Conclusion

Cette thèse a étudié la distribution de l'information sur l'Internet. Dans ce chapitre nous présentons un sommaire de nos contributions et présentons des directions possibles pour des travaux futurs.

Résumé

Dans la première partie de cette thèse, couvrant les chapitres 3 – 5, nous avons étudié la redirection des clients dans des architectures de distribution de contenu. Nous avons présenté le "Location Data System" (LDS) qui permet à des clients de localiser des copies d'objets se trouvant sur différents serveurs d'objets sur le Web. LDS est une extension du Domain Name System (DNS) existant. Quand un client veut localiser un objet, il envoie une requête DNS à l'URL complète et reçoit comme réponse une liste de tous les serveurs d'objets qui ont une copie de l'objet demandé. Le système LDS peut être mis en place d'une manière incrémentale et il peut être modifié afin de réduire le nombre de requêtes envoyées dans le réseau.

Nous avons présenté une architecture pour un service d'annuaire répliqué à l'échelle de l'Internet. Comme exemple de notre architecture, nous avons montré comment le système DNS actuel pourrait être réalisé avec cette architecture répliquée. L'atout principal de cette nouvelle architecture est qu'elle nous permet de stocker des informations qui changent rapidement et de garantir que les données sont cohérentes dans tout le réseau. De plus, notre architecture peut réduire considérablement le temps de latence des requêtes DNS. Notre architecture peut être déployée d'une manière incrémentale et une de ses qualités principales est qu'elle n'exige aucune modification des logiciels DNS existants, ni aucun changement de syntaxe ou de sémantique des messages DNS. Nous avons également effectué une évaluation des performances pour déterminer des valeurs appropriées de la durée de temps pendant laquelle nous pouvons cacher l'information dans les serveurs de noms.

En tant que notre troisième contribution dans la première partie de cette thèse, nous avons effectué une évaluation numérique de la performance des mécanismes de redirection de client employés par les réseaux modernes de distribution de contenu. Les CDNs actuels utilisent le DNS pour rediriger les clients, soit suivant la redirection totale ou la redirection sélective. La redirection sélective nécessite l'ouverture de nouvelles connexions TCP et, en utilisant des simulations, nous avons mesuré le coût associé à ces nouvelles connexions. Nos résultats indiquent que ce coût est en général très élevé. Nous avons également vérifié ces résultats de simulation avec des expériences sur des serveurs de l'Internet et nous avons observé des résultats semblables à nos simulations.

Dans la deuxième partie de cette thèse, couvrant les chapitres 6 – 8, nous avons étudié la réplication des objets dans des architectures de distribution de contenu. Nous avons étudié des stratégies de réplication d'objets dans les réseaux de distribution de contenu (CDN). Nous avons formulé le problème de placement optimal des objets comme un problème d'optimisation combinatoire et nous avons montré que ce problème est NP-complet. Nous avons développé plusieurs heuristiques différentes pour placer des objets dans un CDN et nous avons évalué leur performances sur de vraies topologies des systèmes autonomes (AS) de l'Internet. Nous avons trouvé que les heuristiques qui tiennent compte de l'état entier du réseau à travers tous les AS ont une performance supérieure aux heuristiques qui basent leurs décisions seulement sur l'information disponible localement dans un AS. Nous avons également considéré la coopération dans des réseaux de type "peer-to-peer". Nous avons développé un modèle pour évaluer les avantages de la coopération entre les utilisateurs de réseaux peer-to-peer dans les différent AS. Notre modèle fournit également un test simple pour vérifier si la coopération serait avantageuse. Nos résultats ont prouvé que la coopération entre AS voisins rapporte des avantages élevés, mais ces avantages diminuent quand la distance entre les deux AS augmente.

Comme deuxième contribution dans la deuxième partie de la thèse, nous avons considéré la réplication optimale de contenu dans les communautés peer-to-peer. Nous avons distingué deux types de communautés : les caches de contenu et les clubs de contenu. Nous avons modélisé le problème de réplication optimale de contenu dans une communauté comme un problème de programmation de nombre entier. La solution à ce problème nous fournit une borne supérieure au taux de succès (ou "hit") qui peut être obtenu. Nous avons développé des algorithmes adaptatifs qui répliquent des objets dynamiquement et avons comparé les taux de "hit" à la borne supérieure. Nous avons constaté que nos algorithmes, combinés avec une politique de remplacement de type LFU distribué nous donnent des taux de "hit" et des profils de réplication qui sont quasi-optimaux. Nous avons également présenté une formulation du problème initial qui tient compte des "hot-spots" et équilibre la charge à travers tous les nœuds.

Comme contribution finale de cette thèse, nous avons étudié la distribution de vidéos encodées en couche et distribuées à travers des caches. Nous avons considéré le problème avec deux ressources, la mémoire de cache et la capacité de lien de réseau, et nous l'avons modélisé en utilisant un modèle de knapsack stochastique. Nous avons proposé trois heuristiques pour déterminer quelles couches de quelles vidéos doivent être cachées pour maximiser le revenu accru. Grâce à des expériences numériques nous avons constaté que toutes nos heuristiques offrent de bonnes performances et que la meilleure performance est obtenue avec l'heuristique dite "revenue density". Un opérateur de cache pourrait utiliser ces heuristiques pour déterminer dynamiquement l'utilité de cacher une couche donnée d'une vidéo donnée. Nous avons également considéré deux extensions intuitives au modèle initial : la renégociation de qualité et une file d'attente pour les requêtes. Nos résultats indiquent que ces extensions ne fournissent pas de gain significatif dans le revenu global. De plus, nous avons considéré le problème où des clients sont intéressés seulement par des vidéos complètes. Nous avons trouvé que dans ce cas particulier, il n'est pas avantageux de cacher seulement quelques couches d'une vidéo et que nous devrions cacher toutes les couches d'une même vidéo ensemble.

Orientations pour des travaux futurs

Cette étude donne lieu à plusieurs orientations possibles pour des travaux futurs. Chacun des chapitres présente des possibilités de continuation des travaux du chapitre. La distribution de l'information, et surtout les réseaux peer-to-peer, sont des domaines très prometteurs pour la recherche future. Nous présentons maintenant comment les travaux de cette thèse pourraient être étendus dans cette direction.

Les nouveaux réseaux, tels que les réseaux "ad-hoc" sans fil et mobiles, sont appropriés à une distribution de contenu de type peer-to-peer. La nature éphémère de ces réseaux exigera de nouveaux protocoles et de nouvelles architectures pour assurer une distribution de contenu efficace, aussi bien du contenu statique que du contenu 'streamé'. Ces nouveaux réseaux sont susceptibles de devenir très populaires et ils méritent des études approfondies. Accompagnant cette recherche, il est aussi important de développer des prototypes pour tester les nouveaux protocoles et les nouvelles architectures dans la pratique.

Quelques problèmes clés dans de tels réseaux sont la sécurité, la protection de la vie privée, et le nommage des objets. Les réseaux actuels peer-to-peer n'offrent aucune protection contre des utilisateurs malveillants essayant d'injecter de faux contenus ou de refuser le service aux autres utilisateurs. Dans un environnement où les utilisateurs sont fortement liés l'un à l'autre, tel que dans un réseau ad-hoc sans fil, ces problèmes doivent être résolus.

En outre, dans un réseau où tous les nœuds participent à son fonctionnement, nous devons nous assurer que la protection de la vie privée des utilisateurs est assurée. La manière d'atteindre ce but peut dépendre de l'application ou des préférences de l'utilisateur. Quelques utilisateurs peuvent même souhaiter rester complètement anonymes et le réseau devrait offrir cette possibilité.

Nous avons également besoin de méthodes efficaces pour trouver des objets dans ces réseaux. Ceci exige un système de nommage efficace mais aussi des méthodes pour trouver le contenu. Dans un réseau où le contenu passe librement d'un nœud à l'autre, des systèmes de nommage liant le nom d'un objet à une location, tel que les URLs, sont inadéquats. De nouveaux systèmes de nommage doivent être développés et leur performances doivent être évalués. Les mécanismes de requête pour trouver le contenu sont étroitement liées au système de nommage et leur performance est critique à la performance globale du système. Par conséquent, ils doivent être étudiés d'une manière approfondie afin d'éviter qu'ils deviennent le goulot d'étranglement du système.

Bibliography

[1] E. Adar and B. A. Huberman. Free riding on Gnutella. Technical report, Internet Ecologies Area, Xerox Palo Alto Research Center, September 2000.

[2] Adero.
<http://www.adero.com/>.

[3] Akamai.
<http://www.akamai.com>.

[4] P. Albitz and C. Liu. *DNS and BIND*. O'Reilly & Associates, Inc., 1994.

[5] Y. Amir, A. Peterson, and D. Shaw. Seamlessly selecting the best copy from internet-wide replicated web servers. In *Proceedings of 12th International Symposium on Distributed Computing (DISC'98)*, Andros, Greece, September 1998.

[6] The Apache Software Foundation.
<http://www.apache.org/>.

[7] S. M. Baker and B. Moon. Distributed cooperative web servers. In *Proceedings of 8th International World Wide Web Conference*, Toronto, Canada, May 11–14, 1999.

[8] T. Berners-Lee, L. Masinter, and M. McCahill. *RFC1738: Uniform Resource Locators (URL)*, December 1994.

[9] Y. Birk. Random RAIDs with selective exploitation of redundancy for high performance video servers. In *Proceedings of NOSSDAV*, St. Louis, MO, May 1997.

[10] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom*, New York, NY, March 21–25, 1999.

[11] Chronicle of Higher Education. Napster was nothing compared with this year's bandwidth problems, September 28, 2001. Available also at,
<http://chronicle.com/free/v48/i05/05a04401.htm>.

[12] I. Cidon, S. Kutten, and R. Soffer. Optimal allocation of electronic content. In *Proceedings of IEEE Infocom*, Anchorage, AK, April 22–26, 2001.

[13] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing web latency. In *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 26–30 2000.

[14] I. Cooper, I. Melve, and G. Tomlinson. *RFC 3040: Internet Web Replication and Caching Taxonomy*, January 2001.

[15] Debian GNU/Linux web site.
`<http://www.debian.org>`.

[16] Digital Island Inc.
`<http://www.digisle.net>`.

[17] DirecPC home page.
`<http://www.direcpc.com/>`.

[18] D. Eastlake. *RFC 2535: Domain Name System Security Extensions*, March 1999.

[19] H. Eriksson. MBONE: the multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[20] Farsite.
`<http://research.microsoft.com/sn/Farsite/>`.

[21] FastTrack.
`<http://www.fasttrack.nu/>`.

[22] R. T. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, June 1999.

[23] Freenet project.
`<http://freenet.sourceforge.net/>`.

[24] S. Gadde, J. Chase, and M. Rabinovich. Directory structures for scalable internet caches. Technical Report CS-1997-18, Department of Computer Science, Duke University, 1997.

[25] L. Gao. On inferring autonomous system relationships in the internet. In *Proceedings of IEEE Global Internet*, San Francisco, CA, November 28–30, 2000.

[26] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[27] D. J. Gemmel, H. M. Vin, D. D. Kandalur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Multimedia*, 28(5):40–49, May 1995.

[28] Gnutella web site.
`<http://www.gnutella.co.uk/>`.

[29] C. Grimm, J.-S. Vöckler, and H. Pralle. Request routing in cache meshes. *Computer Networks and ISDN Systems*, 30(22–23):2269–2278, November 1998.

[30] 100hot sites.
`<http://www.hot100.com/>`.

[31] Internet domain survey, July 1999.
`<http://www.isc.org/ds/>`.

[32] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. 2001. Submitted.

[33] S. Jin and A. Bestavros. GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *Proceedings of 5th Web Caching and Content Distribution Workshop*, Lisbon, Portugal, May 22–24, 2000.

[34] M. Jung, J. Nonnenmacher, and E. W. Biersack. Reliable multicast via satellite: Unidirectional vs. bi-directional communication. In *Proceedings of KiVS'99*, Darmstadt, Germany, March 1999.

[35] J. Kangasharju and K. W. Ross. A replicated architecture for the domain name system. In *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 26–30 2000.

[36] J. Kangasharju, K. W. Ross, and J. W. Roberts. Locating copies of objects using the domain name system. In *Proceedings of 4th Web Caching Workshop*, San Diego, CA, March 31 – April 2, 1999.

[37] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrpahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of 29th Annual ACm Symposium on Theory of Computing*, El Paso, TX, May 1997.

[38] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, October 2000.

[39] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, November 2000.

[40] J. Lee, T. Kim, and S. Ko. Motion prediction based on temporal layering for layered video coding. In *Proceedings of ITC–CSCC*, volume 1, July 1998.

[41] B. Li, M. J. Golin, G. F. Italiano, and X. Deng. On the optimal placement of web proxies in the internet. In *Proceedings of IEEE Infocom*, New York, NY, March 21–25, 1999.

[42] S. McCanne and M. Vetterli. Joint source/channel coding for multicast packet video. In *Proceedings of ICIP*, October 1995.

[43] Mirror Image Internet.
<http://www.mirror-image.com>.

[44] R. Moats. *RFC2141: URN Syntax*, May 1997.

[45] P. V. Mockapetris. *RFC 1034: Domain names — concepts and facilities*, November 1987.

[46] P. V. Mockapetris. *RFC 1035: Domain names — implementation and specification*, November 1987.

[47] Mojo nation.
<http://www.mojonation.net>.

[48] Morpheus.
<http://www.musiccity.com/>.

[49] Msn.
<http://www.msn.com>.

[50] A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the internet. In *Proceedings of IEEE Infocom*, New York, NY, March 21–25, 1999.

[51] Napster.
<http://www.napster.com>.

[52] A distributed testbed for national information provisioning.
<http://ircache.nlanr.net/>.

[53] Netcraft web server survey.
<http://www.netcraft.com/survey/>.

[54] Netscape.
<http://www.netscape.com>.

[55] UCB/LBNL/VINT network simulator - ns (version 2).
<http://www-mash.cs.berkeley.edu/ns/>.

[56] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proceedings of ACM SIGCOMM*, Cannes, France, September 14–18 1997.

[57] NLANR measurement and operations analysis team.
<http://moat.nlanr.net>.

[58] J. Nonnenmacher, E. W. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast. *IEEE/ACM Transactions on Networking*, 6(4):349–361, August 1998.

[59] V. N. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 28 – September 1, 2000.

[60] H. Peyravi. Medium access control protocols performance in satellite communications. *IEEE Communications Magazine*, 37(3):62–71, March 1999.

[61] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *Proceedings of IEEE Infocom*, Anchorage, AK, April 22–26, 2001.

[62] M. Rabinovich and A. Aggarwal. RaDaR: A scalable architecture for a global web hosting service. In *Proceedings of 8th International World Wide Web Conference*, Toronto, Canada, May 11–14, 1999.

[63] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, San Diego, CA, August 27–31, 2001.

[64] M. Reisslein, F. Hartanto, and K. W. Ross. Interactive video streaming with proxy servers. In *Proceedings of First International Workshop on Intelligent Multimedia Computing and Networking*, Atlantic City, NJ, February 2000.

[65] R. Rejaie, M. Handley, and D. Estrin. Quality adaptation for congestion controlled video playback over the internet. In *Proceedings of ACM SIGCOMM*, Cambridge, MA, August 30 – September 3, 1999.

[66] 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*, New York, NY, March 1999.

[67] R. Rejaie, H. Yu, M. Handley, and D. Estrin. Multimedia proxy caching mechanism for quality adaptive streaming applications in the internet. In *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 26–30, 2000.

[68] Réseau national de télécommuncations pour la technologie, l'enseignement et la recherche.
<`http://www.renater.fr`>.

[69] P. Rodriguez, A. Kirpal, and E. W. Biersack. Parallel-access for mirror sites in the internet. In *Proceedings of IEEE Infocom*, Tel Aviv, Israel, March 26–30 2000.

[70] P. Rodriguez, K. W. Ross, and E. W. Biersack. Improving the WWW: Caching or multicast? *Computer Networks and ISDN Systems*, 30(22–23):2223–2243, November 1998.

[71] DNS root servers.
<ftp://rs.internic.net/netinfo/root-servers.txt>.

[72] K. W. Ross. *Multiservice Loss Models for Broadband Telecommunication Networks.* Springer–Verlag, 1995.

[73] A. Rousskov and D. Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22–23):2155–2168, November 1998.

[74] Route views project homepage.
<http://www.antc.uoregon.edu/route-views/>.

[75] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[76] Sandpiper.
<http://www.sandpiper.com>.

[77] H. Schulzrinne, A. Rao, and R. Lanphier. *RFC 2326: Real Time Streaming Protocol (RTSP)*, April 1998.

[78] S. Sen, J. Rexford, and D. Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of IEEE Infocom*, New York, NY, March 1999.

[79] A. Shaikh, R. Tewari, and M. Agrawal. On the effectiveness of DNS-based server selection. In *Proceedings of IEEE Infocom*, Anchorage, AK, April 22–26, 2001.

[80] SkyCache, inc.
<http://www.skycache.com>.

[81] Squid Internet Object Cache.
<http://www.squid-cache.org/>.

[82] K. Sripanidkulchai. The popularity of Gnutella queries and its implications on scalability, March 2001. Unpublished white paper,
<http://www.cs.cmu.edu/~kunwadee/research/p2p/gnutella.html>.

[83] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, San Diego, CA, August 27–31, 2001.

[84] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[85] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, Department of Computer Sciences, University of Texas at Austin, 1998.

[86] M. Vishwanath and P. Chou. An efficient algorithm for hierarchical compression of video. In *Proceedings of ICIP*, November 1994.

[87] P. Vixie, S. Thomson, Y. Rekhter, and J. Bound. *RFC 2136: Dynamic Updates in the Domain Name System (DNS UPDATE)*, April 1997.

[88] Y. Wang, Z. Zhang, D. Du, and D. Su. A network–conscious approach to end-to-end video delivery over wide area networks using proxy servers. In *Proceedings of IEEE Infocom*, San Francisco, CA, April 1998.

[89] Z. Wang and P. Cao. Persistent connection behavior of popular browsers. Unpublished white paper,
<`http://www.cs.wisc.edu/~cao/papers/persistent-connection.html`>.

[90] D. Wessels. *Web Caching*. O'Reilly & Associates, Inc., 2001.

[91] D. Wessels and K. Claffy. *RFC 2186: Internet Cache Protocol (ICP), version 2*, September 1997.

[92] S. Williams, M. Abrams, G. Abdulla, S. Patel, R. Ribler, and E. A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM*, Stanford, CA, August 26–30, 1996.

[93] World Wide Web Consortium. *HTML 4.01 Specification*, December 24, 1999. W3C Recommendation.

[94] Yahoo.
<`http://www.yahoo.com`>.

[95] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California, Berkeley, April 2000.

Annexe A

Résumé de la thèse en français

A.1 Introduction

Ce chapitre résume les contributions de chaque chapitre en français. Pour chaque chapitre, nous présentons les motivations, une sommaire et les résultats principaux.

A.2 Distribution de l'information

Dans le chapitre 2 nous présentons les architectures de distribution de contenu utilisées sur Internet. Le modèle client-serveur traditionnel n'est pas suffisant pour garantir une bonne qualité de service quand la charge sur un serveur devient trop élevée. La première solution est d'utiliser des caches web à côté des clients pour réduire cette charge (Section 2.2). Cette solution, bien qu'efficace, a quand même quelques inconvénients majeurs. Notamment, le fournisseur de contenu n'a aucun contrôle sur les objets stockés dans les caches, ce qui rend difficile la construction des sites web avec un contenu dynamique.

Pour résoudre ce problème, les réseaux de distribution de contenu (content distribution networks, ou CDNs) ont été développés (Section 2.3). Dans ces réseaux, le fournisseur de contenu confie le contenu à un tiers, c'est-à-dire, au distributeur de contenu. Le distributeur a mis en place des serveurs de contenu près des utilisateurs. Ces serveurs de contenu sont similaires aux caches web traditionnels, mais avec une différence importante. Dans les réseaux de distribution le fournisseur de contenu peut contrôler comment son contenu est stocké dans les serveurs de contenu, ce qui permet un contrôle plus effectif sur le contenu dynamique des sites.

Les réseaux de type "peer-to-peer" (Section 2.4) sont une nouvelle forme de distribution de contenu. Dans ces réseaux, chaque utilisateur est en même temps client et serveur. Les utilisateurs fournissent de l'espace disque pour stocker les objets dans le réseau. Pour trouver les objets stockés dans le réseau, il y a un service d'annuaire, qui peut être centralisé ou distribué. Des solutions hybrides sont aussi apparues récemment.

4	h	t	m	l	5	i	n	d	e	x	4	~	b	o	b	4	h	t
t	p	3	w	w	w	7	e	u	r	e	c	o	m	2	f	r	0	

Fig. A.1 – Codage d'URL en DNS

A.3 Localisation des objets

Dans le chapitre 3, nous présentons notre architecture pour localiser des copies d'objets qui se trouvent dans les différents serveurs d'objet.

A.3.1 Motivation

Dans le Web actuel, les clients n'ont aucun moyen pour localiser des objets qui peuvent se trouver sur n'importe quel serveur. En particulier, ce problème est présent dans les hiérarchies de cache ou les requêtes sont traitées d'une manière statique. En plus, une hiérarchie se termine par la racine et ne peut s'étendre plus loin. Ceci peut être un problème parce qu'il peut y avoir un cache rapide contenant l'objet désiré près du serveur d'origine qui, à son tour, peut être surchargé ou posséder une connexion très lente. Ce sont ces problèmes dans l'infrastructure actuelle qui nous ont motivés dans notre travail.

A.3.2 Location Data System (LDS)

Dans cette section, nous présentons une résumé de notre solution que nous avons baptisé le "Location Data System" ou "LDS". Le LDS peut être vu comme une boîte noire qui fournit le service de localisation aux clients. La boîte noire prend l'URL fourni par le client et retourne une liste contenant les adresses IP de tous les serveurs possédant une copie de cet URL. Au lieu de donner l'URL complet à la boîte noire, le client peut aussi utiliser un préfixe de l'URL ; dans le cas limite, ce préfixe se réduit au nom de la machine et, dans ce cas, le service LDS est identique au service DNS (Domain Name System).

Étant donné les similarités entre LDS et DNS, nous avons décidé de baser le LDS sur le DNS. En fait, pour réaliser le service LDS, il suffit d'ajouter un nouveau type de "resource record" (RR) dans le DNS. Chacun de ces nouveaux RRs est associé à un URL et contient les adresses des serveurs possédant une copie de cet URL. Cette modification ne nécessite qu'un travail minimal parce que le DNS contient déjà différents types de RRs. Ajouter un RR supplémentaire dans un serveur de noms ne présentera pas de grandes difficultés. La Figure A.1 montre comment on peut coder l'URL `http ://www.eurecom.fr/~bob/index.html` en utilisant le codage du DNS.

Dans le chapitre 3, nous présentons la manière d'appliquer le LDS dans le contexte de caches Web. Le problème majeur du LDS est la quantité de trafic injectée dans le réseau parce que le client doit envoyer une requête LDS pour chaque URL. Nous proposons quelques solutions possibles à ce problème. Un deuxième question ouverte dans le LDS est de choisir parmi les serveurs possédant l'URL donné. Dans ce chapitre nous offrons quelques solutions possibles.

A.4 Réplication de DNS

Domain Name System (DNS) est une base de données distribuée contenant les informations vitales pour le bon fonctionnement d'Internet. Le Chapitre 4 présente notre architecture pour un DNS répliqué.

A.4.1 Motivation

Le DNS actuel présente deux défauts majeurs. Premièrement, le DNS n'est pas adapté pour stocker de l'information qui change rapidement. Ceci est utile pour résoudre les problèmes liés à la surcharge d'un serveur Web. Par exemple, si la charge sur un serveur devient trop importante, le serveur peut répliquer son contenu sur un autre serveur et noter ce fait dans le DNS. Mais, comme les clients peuvent garder les information obtenues du DNS dans leur cache, ils ne seraient pas informés du fait que le serveur s'est répliqué. Deuxièmement, le temps que prend une requête DNS peut être long parce que le client doit contacter un serveur lointain pour obtenir l'information.

A.4.2 Architecture

Notre architecture est basée sur une base de données répliquée sur plusieurs serveurs que nous appelons “serveur répliqué”. Ces serveurs sont placés de manière à les mettre près des clients (au minimum un serveur répliqué par pays). Ils contiennent chacun toute la base de données du DNS et peuvent donc répondre à toutes les requêtes. Quand un client envoie une requête DNS, il l'enverra au serveur répliqué le plus proche qui lui répondra. Grâce à la proximité du serveur répliqué, le client obtiendra la réponse rapidement.

Afin de maintenir la cohérence dans la base de données, les serveurs répliqués sont connectés soit par multicast, soit par une liaison satellite. Chaque serveur répliqué est responsable d'une partie de la base de données et, quand il y a une modification, le serveur responsable envoie cette modification à tous les autres serveurs.

Dans ce chapitre nous présentons aussi une analyse de performance de notre architecture. Nous étudions les problèmes liés à la cohérence de la base de données et la manière de réduire le trafic en maintenant une cohérence acceptable. Nous faisons aussi quelques expériences sur Internet afin de donner une indication du gain possible en temps pour une requête donnée.

Dans notre évaluation, nous nous concentrons sur le nombre de requêtes envoyées aux serveurs de nom et aussi sur le pourcentage de réponses incorrectes causées par des changements dans les données. Notre évaluation suppose une architecture où le serveur de noms local peut cacher certaines informations reçues d'un serveur répliqué. Cette architecture est montrée dans figure 4.4 (page 71).

La fraction de requêtes envoyées par le serveur local vers le serveur répliqué est

$$\frac{1}{\lambda T + 1} \tag{A.1}$$

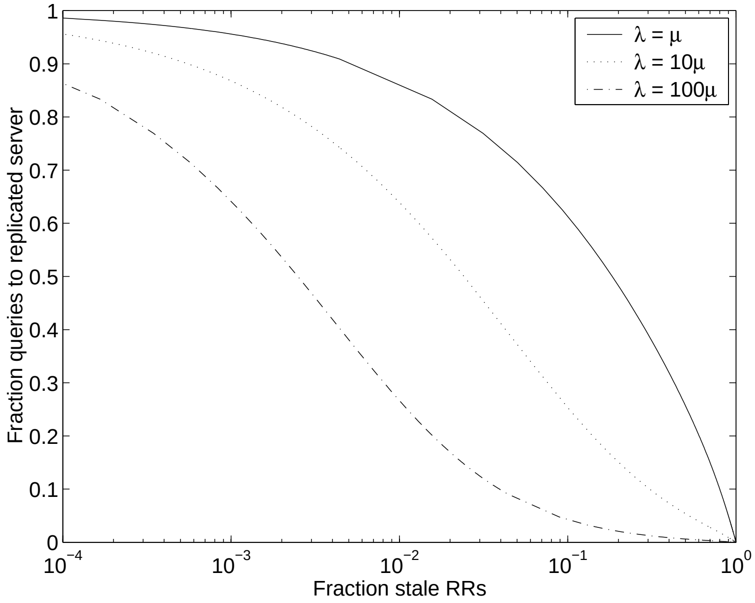


FIG. A.2 – Comparaison de fraction de réponses incorrectes à la fraction de requêtes

où λ est le taux de requêtes et T le temps durant lequel le serveur local peut cacher les informations.

La fraction de réponses incorrectes est

$$\frac{1}{\frac{1}{\lambda}+T}\left(\frac{e^{-\mu T}+\mu T-1}{\mu}\right) \tag{A.2}$$

où μ est le taux de changement des informations.

La Figure A.2 montre une comparaison des fractions de réponses incorrectes et les requêtes envoyées vers le serveur répliqué en fonction de T.

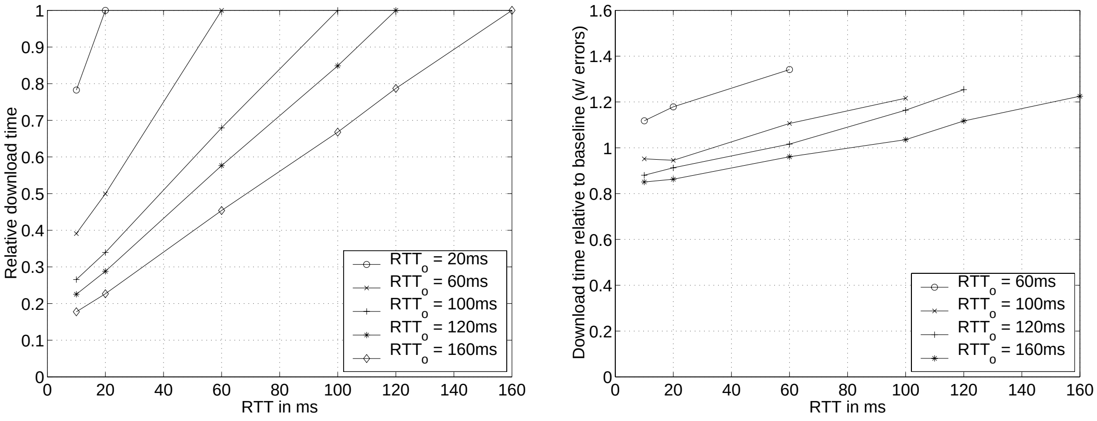
A la fin du chapitre nous montrons comment faire la transition du système DNS actuel vers notre architecture répliquée d'une manière incrémentale. Nous considérons également les questions liées à la sécurité de notre architecture et à la résistance aux pannes.

A.5 Redirection dans les CDNs

Le Chapitre 5 présente nos travaux sur l'évaluation des performances des différentes méthodes de redirection pratiquées par les réseaux de distribution de contenu.

A.5.1 Motivation

Les réseaux de distribution de contenu (CDN) utilisent actuellement des méthodes différentes pour diriger les clients vers les serveurs de contenu. Le coût de cette redirection a une influence directe sur le temps que l'utilisateur doit attendre pour avoir les documents



(a) Rédirection totale

(b) Rédirection sélective

FIG. A.3 – Performance de deux méthodes de redirection

demandés. Nous nous sommes concentrés sur ce problème afin de pouvoir évaluer l'impact de ces méthodes de redirection sur l'utilisateur.

A.5.2 Résultats

En utilisant les simulations et les expériences sur Internet nous avons comparés deux méthodes de redirection actuellement utilisées par les différents CDNs. La première, la redirection sélective, utilisée par Akamai [3], consiste à aller jusqu'au serveur d'origine pour avoir la page HTML, puis à aller sur un serveur de contenu pour avoir les images sur la page. La deuxième méthode, la redirection totale, utilisée par d'autres CDNs (par exemple Adero [2] et Digital Island [16]) et, dans une moindre mesure, par Akamai aussi, consiste à diriger le client sur le serveur de contenu pour tous les objets.

La redirection sélective nécessite l'ouverture d'une nouvelle connexion TCP vers le serveur de contenu. L'ouverture de cette nouvelle connexion peut, dans le pire de cas, prendre plusieurs secondes, qui augmentent le temps d'attente perçu par l'utilisateur. La redirection totale peut bénéficier de la connexion ouverte et d'une fenêtre TCP plus grande, ce qui peut réduire le temps d'attente pour l'utilisateur.

Dans nos simulations et expériences, nous avons constaté que la redirection sélective a, en général, une performance supérieure à celle de la redirection totale. Avec la redirection totale, il est possible de répliquer des pages web de manière à avoir une performance identique à la redirection sélective mais, dans ce cas, le fournisseur de contenu doit choisir, typiquement à la main, les objets à répliquer. La redirection sélective atteint ce but automatiquement.

Figure A.3 montre une comparaison des performances des deux méthodes de redirection. Pour plus de détails sur la figure, voir section 5.5.

A.6 Réplication dans les réseaux CDN

Nous présentons notre travail sur la réplication des objets dans un réseau de distribution de contenu dans le chapitre 6. Ce chapitre contient aussi une étude sur la coopération dans les réseaux "peer-to-peer".

A.6.1 Motivation

Un CDN possède un certain nombre de serveurs de contenu. Les clients envoient des requêtes pour les objet stockés sur ces serveurs. Le problème est de savoir comment répliquer les objets afin d'obtenir la meilleure performance possible. La performance peut être définie de plusieurs façons mais, dans ce chapitre, nous utilisons comme métrique le nombre de "hops" dans le réseaux, ce qui correspond approximativement au temps de chargement d'un objet.

A.6.2 Résultats

Nous modélisons le problème comme un problème d'optimisation combinatoire. Soient I serveurs, chacun avec une capacité S_i et un taux de requête λ_i, $\Lambda = \sum_i \lambda_i$, J objets de tailles b_j et une probabilité de requête p_j. Le nombre de "hops" moyen qu'une requête doit traverser est alors

$$C(\mathbf{x}) = \sum_{i=1}^{I} \sum_{j=1}^{J} s_{ij} d_{ij}(\mathbf{x}) \tag{A.3}$$

où $s_{ij} = \lambda_i p_j / \Lambda$ et $d_{ij}(\mathbf{x})$ est la distance la plus courte de i à une copie d'objet j. La matrice $\mathbf{x}$ est definie par les variables x_{ij}

$$x_{ij} = \begin{cases} 1 & \text{si l'objet } j \text{ est stocké au serveur } i, \\ 0 & \text{sinon.} \end{cases}$$

Le problème est de minimiser $C(\mathbf{x})$ sous les contraintes de stockage

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i \qquad i = 1, \ldots, I$$

Nous démontrons que ce problème est NP-complet. Nous développons quatre heuristiques et évaluons leur performances en utilisant une topologie réelle d'une partie d'Internet. La meilleure performance est obtenue quand la stratégie de réplication utilise tous les serveurs avec une vue globale. Cette stratégie est toujours supérieure aux stratégies qui se limitent à une vue locale.

La Figure A.4 montre les performances de nos heuristiques avec 1000 objets et 549 serveurs.

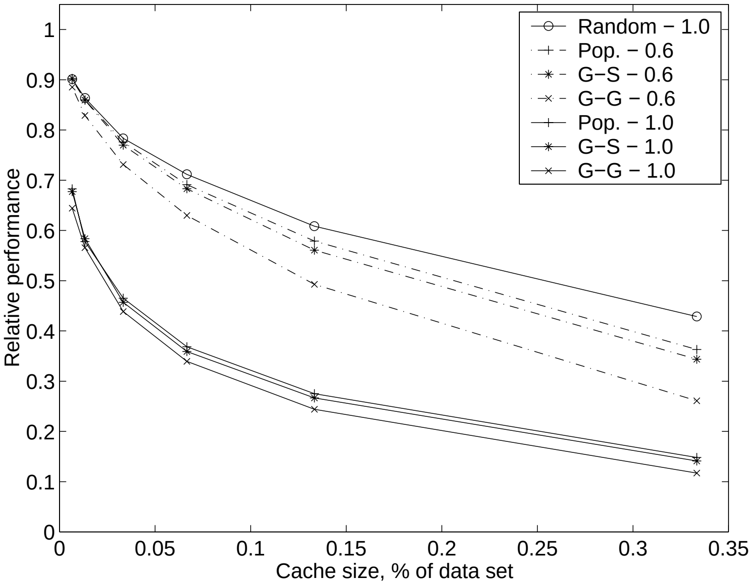


Fig. A.4 – Performance des heuristiques de placement

Nous étudions également la coopération dans les réseaux peer-to-peer, tels que Napster et Gnutella. Nous développons un modèle de coopération. Nous pouvons utiliser ce modèle comme un test pour voir si la coopération est avantageuse. Si

$$\frac{1}{2}\sum_{j=K+1}^{2K-L}(p_j d_{avg} + p_j d_{avg}) - \frac{1}{2}\sum_{j=L+1}^{2K-L} p_j D_{AB} \tag{A.4}$$

est positif, la coopération est avantageuse. La Figure A.5 montre la valeur de (A.4) pour $K = 50$.

Nos résultats montrent que la coopération entre les noeuds dans un réseau peer-to-peer peut considérablement améliorer la performance obtenue par les utilisateurs.

A.7 Les communautés peer-to-peer

Le Chapitre 7 présente la réplication optimale de contenu dans les réseaux peer-to-peer.

A.7.1 Motivation

Récemment, les réseaux peer-to-peer sont devenus très populaires. Pour les utilisateurs sur un campus (université ou entreprise), les ressources, la capacité de stockage et du réseau sont souvent utilisées d'une manière sous-optimale. Notre but est de développer des algorithmes pour la réplication des objets et la localisation de ces copies dans une communauté peer-to-peer.

Nous considérons deux cas : Le premier est le cache de contenu et dans ce cas, la communauté est comme un cache partagé. Le deuxième cas est le club de contenu qui

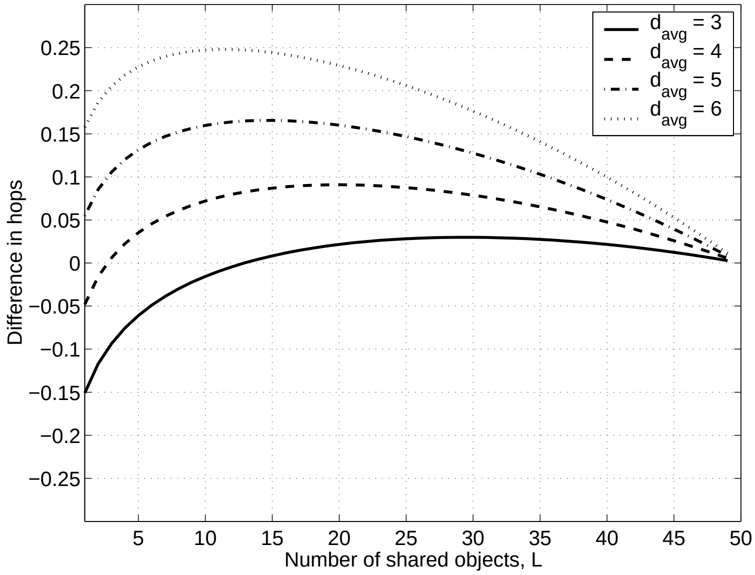


Fig. A.5 – Avantages de coopération

fonctionne comme un vidéo club. Le club offre une sélection de contenu qui est répliqué sur les noeuds.

A.7.2 Résultats

Nous développons le problème comme un problème de "integer programming" qui nous donne la solution optimale. La probabilité de trouver un objet dans la communauté est donnée par

$$P(hit) = \sum_{j=1}^{J} q_j (1 - \prod_{i=1}^{I} p_i^{x_{ij}}) \tag{A.5}$$

où q_j est la probabilité de requête d'objet j, p_i la probabilité que le noeud i soit en marche et x_{ij} est 1 si l'objet j est stocké dans noeud i et 0 sinon.

Le but est de maximiser $P(hit)$ sous les contraintes

$$\sum_{j=1}^{J} b_j x_{ij} \leq S_i, \quad i = 1, \dots, I \tag{A.6}$$

Cette solution est "académique" car elle nécessite la connaissance des popularités des objets et les informations sur le comportement des noeuds. Notre algorithme utilise un double hachage et ceci nous donne les stratégies pour la réplication et la localisation.

Notre algorithme est le suivant : Supposons que X veut obtenir objet j.

1. X détermine Y_1, le noeud gagnant pour j.

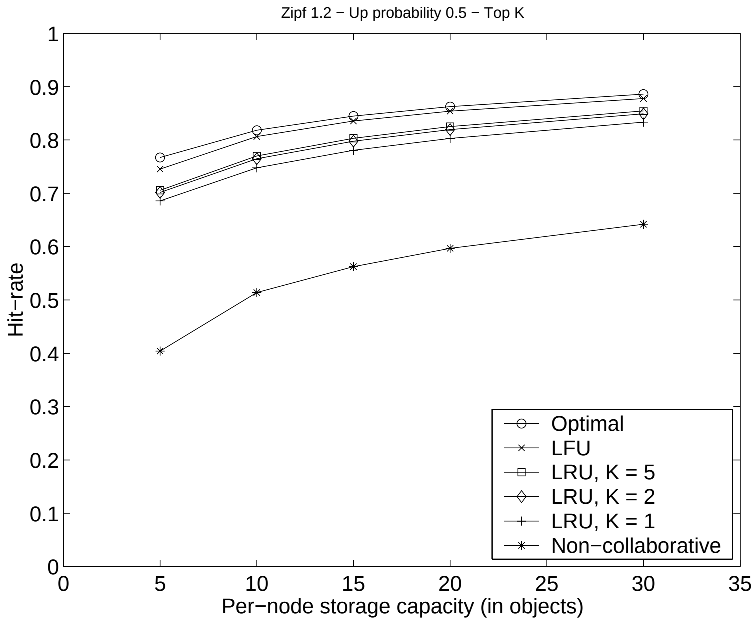


FIG. A.6 – Taux de hit

2. X envoie la requête à Y_1.
   - Si Y_1 a j, il le retourne à X.
   - Si Y_1 n'a pas j, Y_1 détermine les gagnants suivants, $Y_2, \ldots, Y_K$, et envoie un "ping" pour savoir si j est dans ces noeuds.
   - Si l'un d'entre eux a une copie de j, Y_1 le prend, l'envoie à X et crée une copie à Y_1. Si j n'est dans aucun de $Y_2, \ldots, Y_K$, Y_1 le cherche en dehors de la communauté et le stocke.

Nous étudions les performances de cet algorithme avec un simulateur et les résultats indiquent qu'en utilisant une politique de remplacement LFU distribué, la performance obtenue est presque optimale. La politique de remplacement LRU offre une moins bonne performance.

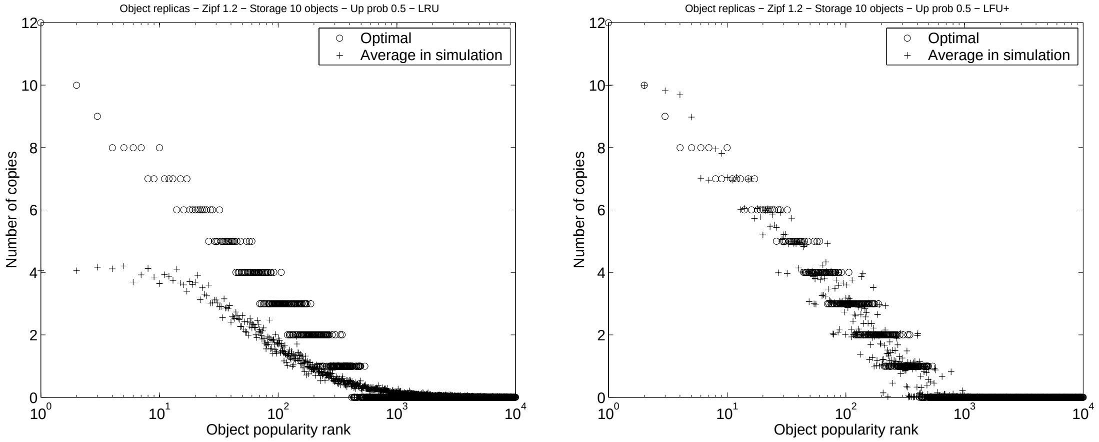
La Figure A.6 montre les taux de hit obtenus par les différentes politiques de remplacement.

La Figure A.7 montre le nombre de copies en fonction de la popularité de l'objet pour LRU, LFU, et le nombre optimal qui maxmimise (A.5).

Dans le cas d'un club de contenu, nous devons décider combien de copies d'un objet nous allons créer quand l'objet est introduit au club. Figure A.8 montre le taux de hit en fonction du nombre de copies initiales (L) et le nombre de gagnants qui sont "pingés" (K).

A.8 Vidéo en couches et les caches

Le Chapitre 8 présente la distribution de la vidéo encodé en couches.



(a) LRU

(b) LFU distribué

Fig. A.7 – Nombre de copies

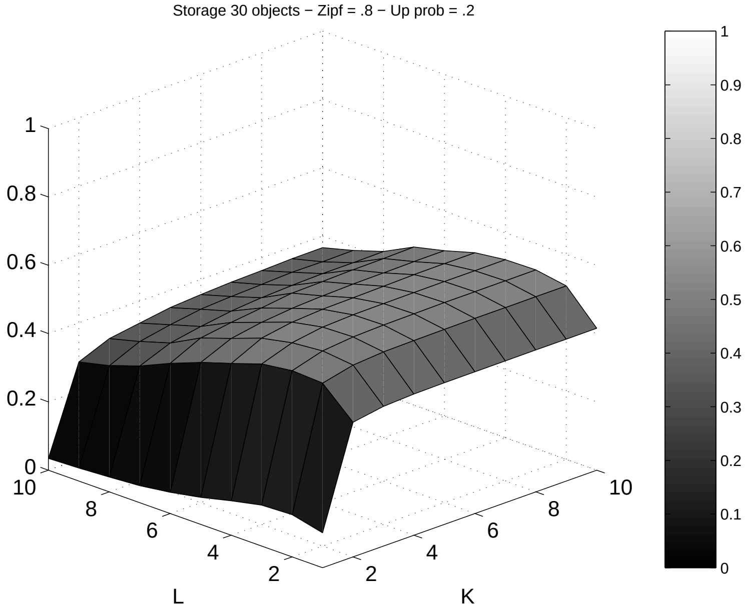
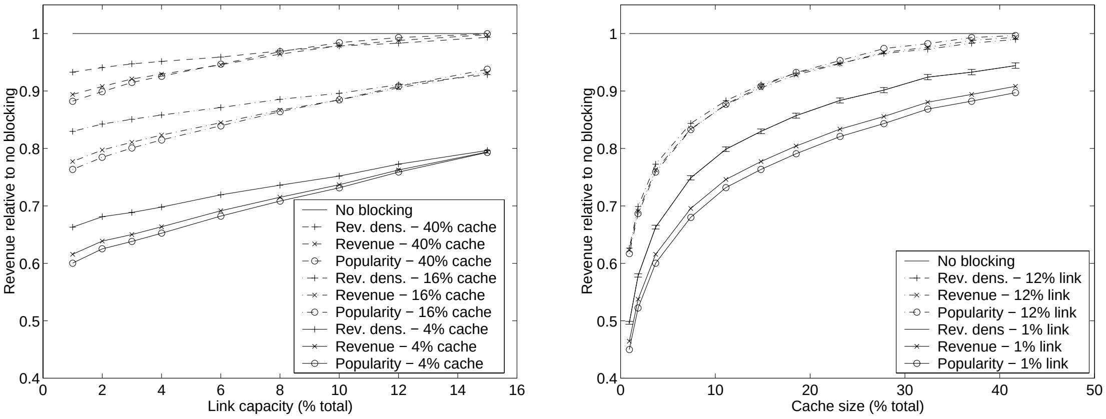


Fig. A.8 – Taux de hit dans un club de contenu

A.8.1 Motivation

Dans un cache web traditionnel, il suffit de décider quels objets doivent être stockés dans le cache pour obtenir la meilleure performance. Pour les objets vidéo encodés en couches, ce problème devient plus compliqué. Nous devons décider non seulement quels objets stocker, mais aussi quelles couches de quelles vidéos nous devons stocker, tout en tenant compte des contraintes sur le décodage, c'est à dire, que les couches basses doivent être présentes pour décoder les couches plus hautes.



(a) En fonction de capacité du lien

(b) En fonction de capacité du cache

FIG. A.9 – Performance des heuristiques

A.8.2 Approche

Nous avons deux ressources, espace de disque dans le cache et capacité du lien entre le cache et le reste du monde, que nous devons utiliser pour maximiser le revenu. Nous modélisons le lien comme un "knapsack" stochastique et nous essayons des stratégies afin de décider quels objets mettre dans le cache.

Le problème revient à maximiser

$$R(\mathbf{c}) = \lambda \sum_{m=1}^{M} \sum_{j=1}^{L} R(j,m)p(j,m)(1 - B_{\mathbf{c}}(j,m)) \tag{A.7}$$

sous la contrainte

$$S(\mathbf{c}) = \sum_{m=1}^{M} \sum_{l=1}^{c_m} r_l(m)T(m) \leq G. \tag{A.8}$$

Pour décider les couches à cacher, nous avons développé des heuristiques parce que le problème ne peut être résolu d'une manière analytique et parce qu'une énumération exhaustive prendrait trop de temps de calcul. Nous développons trois heuristiques et, avec des simulations, nous observons leur performances dans différentes situations.

La Figure A.9 montre les performances de nos heuristiques en fonction de la capacité du lien et celle du cache.

Nous avons aussi considéré la négociation de la qualité, mais nos résultats montrent que la négociation n'apporte que des gains secondaires.